

Eigendom van
HACK42
Hackerspace Arnhem
www.hack42.nl

#42
HACKERSPACE ARNHEM

 **Reflection[®] 2+**

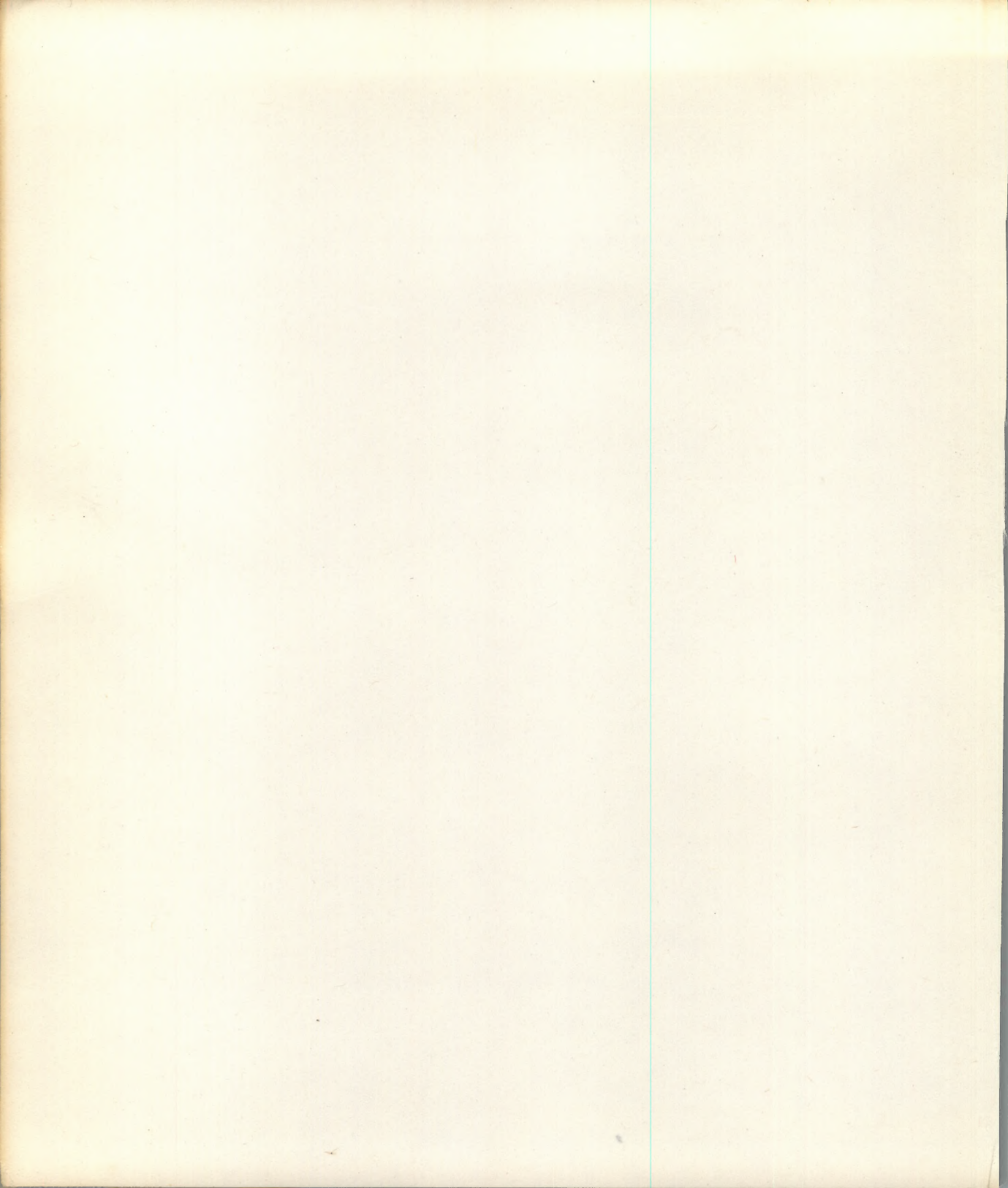
Digital VT320
Terminal Emulation

 **Reflection[®] 4+**

Digital VT340 Terminal
Emulation with ReGIS
and Sixel Graphics

Advanced Topics

▲ D O S



▼
▼
▼
▼

 Reflection® 2+ Digital VT320
Terminal Emulation

 Reflection® 4+ Digital VT340 Terminal
Emulation with ReGIS
and Sixel Graphics

Advanced Topics

▲ D O S

Copyright

© 1992–1994 by Walker Richer & Quinn, Inc. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language, in any form by any means, without the written permission of Walker Richer & Quinn, Inc.

Reflection 2 Plus and Reflection 4 Plus *Advanced Topics*

Version 5.0

May 1994

Licenses and Trademarks

Reflection is a registered trademark of Walker Richer & Quinn, Inc.

Borland International, Inc. — Turbo Pascal is a registered trademark.

Digital Equipment Corporation — DEC, ReGIS, VAX, VMS, VT, VT220, VT240, VT241, VT300, VT330, VT340, and VT420 are trademarks.

Hewlett-Packard Company — Hewlett-Packard and Vectra are registered trademarks, and HP 3000 is a trademark.

International Business Machines Corporation — AT, IBM, and PS/2 are registered trademarks, and PC/XT is a trademark.

Key Tronic Corporation — Key Tronic is a registered trademark.

Microsoft Corporation — Microsoft and MS-DOS are registered trademarks, and QuickBASIC is a trademark.

The Santa Cruz Operation, Inc. — SCO is a registered trademark.

Tektronix, Inc. — TEKTRONIX is a registered trademark.

Unix Systems Laboratories, Inc. — UNIX is a registered trademark.

Customer Service

Walker Richer & Quinn, Inc.

1500 Dexter Avenue North

Seattle WA 98109

USA

Tel: 206-217-7100

Fax: 206-217-0293

European Office

Buitenhof 47

2513 AH Den Haag

The Netherlands

Tel: +31.(0)70.375.11.00

Fax: +31.(0)70.356.12.44

Technical Support in the USA

Technical Support: 206-217-7000

Technical Support Fax: 206-217-9492

Reflection Technical Notes (24-hour automated fax request line): 206-217-9575

Internet Address: support@wrq.com

CompuServe: GO REFLECTION or GO WRQFORUM

Bulletin Board: 206-217-0145

Bulletin Board Telnet Gateway: bbs.wrq.com

Technical Support Outside the USA

Please call your authorized Reflection distributor, or call WRQ for the name of the distributor nearest you.

European Office Bulletin Board: +31.(0)70.356.27.25

This manual is printed on recycled paper with 10% post-consumer waste using soy-based inks.



Table of Contents

Section 1

Keyboard Mapping	1
Introduction	5
Keyboard Mapping Syntax	13
Compiling the Mapping	43
Keyboard Layouts	49
Other Methods of Defining Keystrokes	57

Section 2

VT340 Graphics	63
Using ReGIS Graphics	69
Introduction to ReGIS Commands	73
Position Command	79
Vector Command	83
Curve Command	87
Screen Command	93
Write Command	103
Polygon Fill Command	117
Text Command	121
Load Command	133
Report Command	137
Macrograph Command	143

Section 3

Tektronix 4014 Graphics	145
Using Tektronix Graphics	147

Section 4

Application Program Interface	157
API Support Files	161
Summary of API Function Calls	165
API Function Calls	169
API Error Codes	259

How to Write an API Application	261
Converting Command Language to API	273
Index	281



Figures

Keyboard Mapping and Help on Custom Keys	11
IBM PC or XT Key Names	50
IBM PC/AT Key Names	51
IBM Enhanced 101/102 Key Names	52
Digital LK250 Key Names	53
HP Vectra (non-Enhanced) Key Names	54
Key Tronic 5151 Key Names	55
Coordinate System for ReGIS Screen	75
Pixel Vector Directions	78
Position Command—Cursor Positioning	79
Vector Command—Using Pixel Vectors	84
Curve Command—Center at Current Position	87
Curve Command—Center at Specified Position	88
Curve Command—Arc Center Specified	89
Curve Command—Closed Curve Sequence	90
Curve Command—Open Curve Sequences	91
Write Command—Standard Line Patterns	104
Write Command—Shading Options	112
Write Command—Overlay Writing	113
Write Command—Replace Writing	113
Write Command—Complement Writing	114
Write Command—Erase Writing	114
Polygon Fill Command—Vector Option	117
Polygon Fill Command—Position Option	119
Sample ASCII Characters: 8 x 20 Character Cell	121
Text Command—Italicized Text	130
Text Command—Temporary Text Control	131
Load Command—Designing a Character Cell	136
Load Command—Displaying a Character Cell	136



Tables

Numeric Keypad and Required Keyboard States	23
Mappable Keyboard Functions	27
Control Code Representations	29
VT Control Character Representations	33
Reflection Keyboard Functions	35
VT Terminal Keyboard Functions	36
ReGIS Graphics Keyboard Functions	39
Tektronix Graphics Keyboard Functions	40
Softkeys Graphics Keyboard Functions	40
ANSI Terminal Keyboard Functions	40
VT240 and VT330 Default Monochrome Mappings	96
VT241 Default Color Map	99
VT340 Default Color Map	99
Input Cursor Styles	102
VT240, VT241, and VT330 Writing Planes	116
VT340 Writing Planes	116
7-Bit Character Set Designators	124
National Replacement Character Sets	124
8-Bit Character Set Designators	124
Text Command: Standard Character Cell Sizes	126
Load Command: Hex Codes	135
R Command: Error Conditions	138
Assembly Language Interface	169

Keyboard Mapping

This section provides information on *keyboard mapping*. Keyboard mapping allows you to change the arrangement of Reflection and terminal emulation functions on your keyboard, and to set up keystrokes to automate frequently typed character sequences.

The following chapters are included in this section:

Chapter 1, *Introduction*, briefly outlines the steps involved in creating a keyboard mapping. It is for users who want a quick reference to syntax and a few samples to work from. Mapping files supplied with Reflection are also described here.

Chapter 2, *Keyboard Mapping Syntax*, explains the keyboard mapping syntax in detail: how to identify your keyboard, how to indicate whether you are using Reflection's ANSI emulation, and how to set the initial state of `[NumLock]` and `[CapsLock]`. Each keystroke definition consists of an input specification (the keystroke) and an output specification (the resulting action), and can include comments.

Chapter 3, *Compiling the Mapping*, describes how to compile the new keystroke definitions into a configuration file. This chapter also explains how to reset Reflection to its default mapping.

Chapter 4, *Keyboard Layouts*, contains the keyboard layout for each of the keyboard-ID choices. Each key has a name that must be used in the keystroke definitions.

Chapter 5, *Other Methods of Defining Keystrokes*, is useful if you have an unusual keyboard and you need to define a keystroke in terms of the scan code it generates.

Chapter 1

Introduction	5
Keyboard Mapping Procedure: Quick Overview	6
Sample Keystroke Definitions	6
Supplied Mapping Files	8
Mapping VT Functions During Installation	8
Mapping for WordPerfect	9
Mapping for Lotus 1-2-3 for VAX/VMS	9
Default Mapping	10
Reflection Help System	10
Keyboard Mapping Syntax: Quick Reference	12

Chapter 2

Keyboard Mapping Syntax	13
Identify your Keyboard	13
Enhanced and LK450	14
PC	14
AT	15
LK250	15
HP-Vectra	16
KB-5151	16
NONENH—Non-Enhanced Keyboard	17
Identify your Terminal Type	17
Set the Initial State of NumLock and CapsLock	18
Keystroke Definition: Elements	19
Keystroke Definition: Input Specification	19
Shift Keys	20
Key Names	21
Special Default Keystrokes	24
Keystroke Definition: Output Specification	26
NULL	27
Keyboard Functions	27
Character Strings	28
ASCII Decimal Values	31
Reflection Commands	32
VT Control Functions	32

Keystroke Definition: Comment	34
Reflection's Default Keyboard Mapping	34

Chapter 3

Compiling the Mapping	43
KEYCOMP Syntax	43
Returning to Default Configuration Values	44
KEYCOMP Error Messages	44

Chapter 4

Keyboard Layouts	49
IBM PC or XT Key Names	50
IBM PC/AT Key Names	51
IBM Enhanced 101/102 Key Names	52
Digital LK250 Key Names	53
HP Vectra (non-Enhanced) Key Names	54
Key Tronic 5151 Key Names	55

Chapter 5

Other Methods of Defining Keystrokes	57
Keyboard Diagnostics	57
Determining Interrupt 9 and 16 Values	58
Using Interrupt 9	59
<Shiftkeys>	59
<Number>	59
HIDE	60
Prefix Options	60
Using Interrupt 16	60
<Shiftkeys>	60
<Number>	61

Introduction

Reflection comes with a default keyboard mapping that provides keystrokes to perform a variety of keyboard functions. These functions include:

- ▲ Reflection functions
- ▲ VT terminal emulation functions
- ▲ ANSI terminal emulation functions

For a listing of Reflection and terminal emulation keyboard functions, and the default keystrokes that perform these functions, see pages 34 through 41 in this manual.

Most keyboard functions can be mapped to different (non-default) keystrokes using Reflection's keyboard mapping procedures. Keyboard mapping also allows you to set up keystrokes that reproduce character strings. With keyboard mapping, you can set up a keystroke that will perform a sequence of up to 127 individual characters and/or keyboard functions.

Keyboard mapping information is compiled into a Reflection configuration file. You can change your keyboard mapping for a different host computer or host application simply by loading a different configuration file.

Reflection, by default, only uses your keyboard mapping when you are in "terminal mode," interacting with the host computer. Standard PC keystrokes are in effect when you are using the following Reflection functions:

- ▲ The menu bar or a menu
- ▲ Any dialog box (including the File Transfer, Modem Dialer, and Help dialog boxes)
- ▲ The command line

Use the Reflection command `SET GLOBAL-REMAPPING YES` to bring your keyboard mapping into effect at all times. (The Reflection default is `SET GLOBAL-REMAPPING NO`.) If you enter this command on the command line, you must exit the command line once before you can use your keyboard mapping *at* the command line.

Keyboard Mapping Procedure: Quick Overview

The steps involved in creating a new keyboard mapping are as follows:

1. Use a text editor to create a standard ASCII file.
2. The first line in the file identifies your keyboard.
3. The second line in the file indicates your terminal (VT or ANSI). If this step is skipped, a VT terminal is assumed.
4. The next two lines in the file can indicate the initial state of NumLock and CapsLock (activated or deactivated). If these lines are omitted, these two keyboard states do not change when you load a mapping.
5. Add definitions to your mapping file to map functions to particular keystrokes. Any keys not specifically remapped retain their default Reflection values.
6. After you save and exit your mapping file, use the KEYCOMP utility to compile it into a designated configuration file (R2.CFG or R4.CFG by default).

Sample Keystroke Definitions

The samples below show typical keystroke definitions. The mapping file with the definitions must be compiled into a configuration file that is used with Reflection before the new keystrokes take effect. See "Compiling the Mapping," starting on page 43, for more information.

The following sample maps an Enhanced keyboard so that its numeric and middle keypads function like those on a VT keyboard:

```
keyboard-ID = enhanced
num-lock = vt-pf1           ; use shift num-lock to toggle numlock
kp-slash = vt-pf2
kp-star = vt-pf3
kp-minus = vt-pf4
kp-enter = vt-enter
kp-plus = vt-minus
shift kp-plus = vt-comma
print-screen = vt-f15       ; HELP
scroll-lock = vt-f16        ; DO
pause = hold-screen         ; (ctrl pause is host-break)
```


In the following examples, *shift states* are described by `alt`, `ctrl`, and `shift ctrl`. The *key names* are `home`, `h`, `x`, and `end`. The desired result of the key mapping appears after the equal sign.

`(Alt)-(Home)` sends a disconnect signal:

```
alt home = disconnect
```

`(Ctrl)-(H)` does a login with a password, each followed by a carriage return:

```
ctrl h = "myname ^M password ^M"
```

The keystroke to exit Reflection (`(Alt)-(X)`) is disabled—pressing it has no effect:

```
alt x = null
```

`(Shift)-(Ctrl)-(End)` toggles printer logging as described for VT300 and VT400 series emulation:

```
shift ctrl end = toggle-print-logging
```

The following example demonstrates how the output specification for a keystroke can be made up of several different elements, separated by spaces. Here's what happens each time the keystroke `(Alt)-(→)` (cursor key) is pressed:

1. Go to the Reflection command line.
2. Issue the VT control function for setting the screen display to inverse video.
3. Send a carriage return (^M) to execute the Reflection command, then send a second carriage return to leave the command line.

There are two ways to present the control sequence introducer (^{C_{SI}}) used in this output specification: as a 7-bit control character (`^[[]`) or as an ASCII decimal value (`^(155)`). See "Using Control Functions" in the *Technical Reference* for a table of C0 and C1 controls and their decimal equivalents.

```
alt cp-right = command-line "display '^[[]?5h' ^M^M"
```

Some UNIX host programs use the control character for "negative acknowledge" (NAK) to invoke a function. The default keystroke for generating a NAK is `(Ctrl)-(U)`. To assign NAK to a different keystroke—in this case, `(Ctrl)-(Shift)-(Z)`—use the following line in your keyboard mapping file:

```
ctrl shift z = "^u"
```


Supplied Mapping Files

If you would like to change the way your keyboard is mapped, the easiest (and recommended) method is to use a *supplied* keyboard mapping file. Several source files for defining alternative keyboard mappings are provided on the Reflection disks. You can make copies of these files and use them to remap your keyboard, or just look at them for examples.

The “.KBM” extension identifies keyboard files. If you create your own files, this extension is recommended.

Mapping VT Functions During Installation

In Reflection's installation program, you can opt to remap VT functions by responding to a series of prompts. After installation, type `KEYCOMP /S` at the DOS prompt to be prompted through the same process.

There is a supplied mapping file for each of the most common keyboards. The mapping is intended to make the PC keyboard simulate a VT100 or VT300 series keyboard. The filename is an abbreviation of the corresponding keyboard type followed by a sequence number.

Use the table below to choose a file:

Filename	Keyboard type
ENH3.KBM	Enhanced
AT1.KBM	AT
PC1.KBM	PC/XT
VEC1.KBM	HP-Vectra
KB1.KBM	KB-5151
LK250.KBM	Digital's LK250
LK450.KBM	Digital's LK450
NOTENH.KBM	Non-Enhanced

For example, if you have an IBM AT with the Enhanced 101/102 keyboard, you should select ENH3.KBM as your preferred mapping. Copy the file to another filename and work with the copy—this way you can always compare your version against the original if you make changes to the file. It can be viewed and changed in any PC text editor that can create ASCII text files.

Mapping for WordPerfect

The keyboard mapping file WORDPERF.KBM is designed for use with WordPerfect 5.0 on the VAX and an Enhanced keyboard. The keyboard mapping will be familiar to users who have learned WordPerfect on the PC; keyboard function is almost identical.

See the comments at the beginning of WORDPERF.KBM for details on how to use the mapping file. If you are using a version of WordPerfect that predates version 5.0, or if you are not using an Enhanced keyboard, check the Reflection bulletin board service for other mapping files. The BBS number is listed on page ii of this manual.

Mapping for Lotus 1-2-3 for VAX/VMS

The keyboard mapping file 123VAX.KBM is designed for users who have learned Lotus 1-2-3 on a PC and want to be able to use the same keystrokes when they run Lotus 1-2-3 for VAX/VMS. The mapping becomes part of your Reflection configuration file and does the keystroke "translation."

This mapping file is intended for use on a PC with an Enhanced keyboard. To keep your Lotus mapping and your standard mapping separate, follow these steps:

1. Copy your R2.CFG or R4.CFG configuration file to 123VAX.CFG.
2. Use KEYCOMP to compile 123VAX.KBM into 123VAX.CFG.
3. Run Reflection and start 1-2-3 for VAX/VMS.
4. Select Command Line from the Tools menu to display Reflection's command line.
5. Type `LOAD 123VAX.CFG` on the command line and press **Enter**.
6. Use 1-2-3 for VAX/VMS. Remember to ignore the Numlock and ScrollLock lights on your keyboard, observing instead the NUM and SCROLL indicators that Lotus displays in the lower right corner of the screen.
7. After exiting Lotus 1-2-3 for VAX/VMS, you'll want to load your standard keyboard map. Display Reflection's command line, type `LOAD R<n>.CFG` and press **Enter**.

Default Mapping

The file DEFAULT.KBM, on one of the support disks, lists the default keystrokes and function names for Reflection. It is not meant to be compiled: it is for documentation only. It shows you how the keyboard behaves if you do not make use of the keyboard mapping feature. You can use portions of this file as a starting point for your own mapping files. (If you use a part of this file, delete the semi-colon at the beginning of each line; otherwise the line is treated as a comment.)

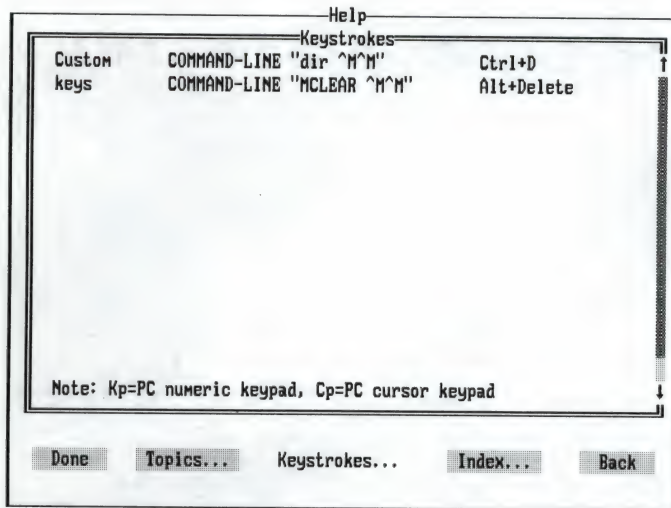
Note: If the file is not in your Reflection directory, copy it from your product disks (type `SETUP /D <drive:>DEFAULT.*` at the DOS prompt to decompress just this file). ▲

Reflection Help System

Reflection's Keystrokes help reflects the current keyboard mapping. When you map a keyboard function to a non-default keystroke, keyboard help displays the new keystroke next to that function. If you create new key definitions, they are displayed as Custom Keys by Keystrokes help.

The keyboard mapping file listed below assigns keystrokes to perform the DIR command from the Reflection command line, and to clear your screen. After this file is compiled into a Reflection configuration file, choose Keystrokes from the Help menu, and select Custom Keys to see the keystrokes you have defined.

```
keyboard-id = Enhanced
term = VT
ctrl d = command-line "dir ^M^M"
alt cp-del = command-line "MCLEAR ^M^M"
```



Keyboard Mapping and Help on Custom Keys

First entry; required

First entry; required

Optional entry for VT terminal; required for ANSI

Optional entry

See below for syntax

IGNORE-SHIFTS is valid only with <int9spec>

IGNORE-SHIFTS is valid only with <int9spec>

NULL disables a keystroke

NULL disables a keystroke

Keyboard Mapping Syntax

This chapter explains the entries that make up a keyboard mapping file. (For a keyboard mapping syntax quick reference, see page 12.)

To create a keyboard mapping file, use a text editor to create a standard ASCII file. You can name this file anything you like, but the filename extension KBM is recommended.

In general, keyboard mapping syntax is *not* case sensitive. Most entries to your keyboard mapping file can be made in uppercase, lowercase, or any combination of the two. There are only two situations where case makes a difference:

- ▲ If you use an alphabetic key name *by itself* in a keystroke definition, case sensitivity applies: `a` is not the same as `A`. (See page 21 for more information.)
- ▲ If you map a keystroke to perform a literal character string, case sensitivity applies: `"HELLO"` is different from `"hello"`.

Identify your Keyboard

The first statement of a keyboard mapping definition source file must be the keyboard-ID statement. This statement defines the keyboard on which the mapping will be used. There must be a space before and after the equal sign.

Each of the following is a valid keyboard-ID statement:

```
keyboard-ID = Enhanced ; IBM 101/102-key keyboard with 12 function
                        ; keys; also, a Digital LK450 keyboard
keyboard-ID = PC       ; IBM PC or PC/XT
keyboard-ID = AT       ; pre-Enhanced IBM AT (ten function keys)
keyboard-ID = LK250    ; Digital LK250 keyboard
keyboard-ID = HP-Vectra ; Hewlett-Packard Vectra (non-Enhanced)
keyboard-ID = KB-5151  ; Key Tronic KB 5151 keyboard
keyboard-ID = NONENH   ; Enhanced keyboard with non-Enhanced BIOS
```

If your keyboard is not among these, choose the option that your keyboard most closely resembles.

Enhanced and LK450

The Enhanced keyboard differs quite a bit from the keyboards that preceded it. It has 12 function keys (rather than 10) and includes these new keys: **Enter**, **/** (kp-slash), and a set of ten cursor control keys. These new keys can be distinguished from the keys on the keypad by PC applications. The key names are shown in the figure on page 52.

Keyboard-ID should also be set to Enhanced for the LK450, a keyboard made by Digital Equipment Corporation. Use the keyboard mapping file LK450.KBM on your product disk if you have this keyboard. Its layout is very similar to that of an Enhanced keyboard, with some keys added:

- ▲ A **Compose** key; its function is mapped in LK450.KBM via its scan code.
- ▲ In LK450.KBM, the `vt-comma` function is mapped to the **,** key, and `vt-minus` is mapped (using its scan code) to the **-** key.
- ▲ **F13** through **F17** have been added; they are mapped via scan codes in LK450.KBM.

The middle cursor pad on the LK450 and Enhanced keyboards share the same key names but are arranged differently. On the LK450:

```
cp-home = find
cp-ins = insert-here
cp-del = remove
cp-end = select
cp-pgup = prev-screen
cp-pgdn = next-screen
```

In Reflection, use the LK450 in IBM compatibility mode (the DEC light should be off; toggle the keyboard mode with **Alt**-**F17**). See page 15 for a description of these keyboard modes.

PC

This is the original PC keyboard, supplied with the IBM PC and PC/XT. The name for each key is shown in the figure on page 50.

AT

This keyboard was on the early IBM AT machines and is nearly identical in appearance and function to the PC keyboard. A few keys have been moved to different positions, including **Esc**, **␣**, and **PrtSc** (kp-star). **SysReq** was added. The name for each key is shown in the figure on page 51.

LK250

The LK250 is a keyboard made by Digital Equipment Corporation. The physical layout of the LK250 matches the VT300 series terminal keyboard. It is basically an AT-style keyboard with several keys added. The layout for the LK250 keyboard is shown on page 53; refer to this figure if you are doing any remapping.

The keys added for the purpose of VT300 series emulation are as follows (*cp* stands for cursor pad, and *kp* stands for keypad):

f11 – f20	select	cp-down
compose	prev	cp-left
find	next	cp-right
insert-here	cp-up	kp-enter
remove		

The LK250 keyboard can be attached to a PC and used in either of two modes, *special* (IBM compatibility mode) or *DEC extended*. When the “special” light at the top of the keyboard is on, the LK250 is in IBM compatibility mode. When it is off, the keyboard is in DEC extended mode. Toggle between these two modes with **Alt-F17**.

DEC Extended Mode

If you are using the LK250.KBM map included on your product diskettes, Reflection toggles the keyboard automatically into DEC extended mode. In this mode, the full functionality of the keyboard is made available and complete VT300 series keyboard emulation is possible.

The keyboard mapping file LK250.KBM can be implemented according to the method described in “Customizing Your Keyboard” in the *User Guide*.

Special Mode

In *special* mode, the LK250 keyboard emulates an old (pre-Enhanced) IBM AT keyboard:

Special Mode

(F11) through (F19)

(Compose)

(F20)

Cursor pad keys

Emulated AT Keystroke

(Shift)-(F1) through (Shift)-(F9)

(Shift)-(F10)

(SysReq)

Unshifted keypad keys

HP-Vectra

The keyboard-ID *HP-Vectra* refers to early HP Vectra keyboards. If you are using the newer HP Vectra keyboard (similar to IBM's Enhanced keyboard), use keyboard-ID = *Enhanced*. See the figure of the Enhanced keyboard on page 52 for the name of each key.

This early Vectra keyboard is very similar to the IBM AT keyboard. HP added a duplicate set of the (F1) through (F8) keys across the top of the keyboard for users accustomed to HP terminals, and an auxiliary set of eleven cursor control keys. Intelligence built into the keyboard attempts to ensure that the cursor and control keys are never interpreted by the PC to generate numeric characters, regardless of the state of (NumLock) and (Shift). The name for each key is shown in the figure on page 54.

KB-5151

This Key Tronic keyboard is another variant of the IBM AT keyboard. There is an extra set of eleven cursor control keys, just as on the HP Vectra. This set of keys cannot be distinguished from the keypad set by PC-based software. It is controlled by a key labeled *Cursr Pad*.

When (CursrPad) is off, the extra cursor keys are deactivated. When it is on, the extra cursor keys are activated and (NumLock) is forced on. An unfortunate side effect of turning on (CursrPad) is that (NumLock) is partially crippled. (CursrPad) is a completely local key—it affects the performance of the keyboard, but is invisible to software.

Neither of the following Key Tronic keys has been assigned a name in Reflection's keyboard mapping language:

- Enter** Because **Enter** cannot be distinguished from **Enter ↵**, it is governed by the mapping for **Enter ↵**.
- Reset** Unless shifted with **Ctrl**, this key does nothing. With **Ctrl** it performs the same function as the keystroke **Ctrl-Alt-Del**; these two keystrokes are indistinguishable and are governed by any mapping for the **Ctrl-Alt-Del** keystroke.

The name for each key is shown in the figure on page 55.

NONENH—Non-Enhanced Keyboard

Use this keyboard-ID if your keyboard is Enhanced (101 key), but your PC is equipped with a non-Enhanced keyboard BIOS. Reflection then interprets keystrokes via their scan codes. Use KEYMON to find out about the BIOS you're using. KEYMON is a program included on your disks for diagnosing keyboard problems. Type **KEYMON** at the DOS prompt.

If you have a non-Enhanced BIOS, you should see the following:

PC keyboard BIOS is standard

In this case, use `keyboard-ID = NONENH` in your mapping definition file. Type **e** to exit KEYMON. For key names, see the figure of the Enhanced keyboard on page 52.

Identify your Terminal Type

You can make Reflection conform to the specifications for SCO ANSI (which is also known as "ANSI console" or "ANSI 80×25"), or use the ANSI settings that are common on electronic bulletin boards. This is done by setting Mode in the General Setup dialog box to either **SCO ANSI** or **BBS ANSI**.

If your terminal mode is set to either **SCO ANSI** or **BBS ANSI**, your default mapping is different than it is for the VT emulations. (You can see how your keyboard is mapped on page 40, or refer to **DEFAULT.KBM**.) To do custom keyboard mapping, your mapping file must contain this line after the keyboard-ID statement:

```
term = ansi
```


Using this statement in your keyboard mapping file doesn't actually *set* your terminal type: that is done by setting the value for Mode in the General Setup dialog box, or by using the Reflection command language equivalent (`SET TERMINAL-TYPE <value>`). Specifying `term = ansi` establishes ANSI for your default keyboard emulation, to which keystroke mapping statements can then be added.

For VT emulation, use `term = vt`. If there is no "term =" statement in a mapping file, a VT terminal-style mapping of the keyboard is assumed.

To use the PC's function keys in ANSI emulation, make sure Reflection is displaying no function key labels by setting Initial Function Keys to None in the Display Setup dialog box. When function key labels are displayed at the bottom of the screen, the function keys behave as indicated on the labels. When the function key labels are not shown, then they perform their ANSI emulation functions.

Set the Initial State of NumLock and CapsLock

The initial state of `NumLock` and `CapsLock` can be defined in your keyboard mapping definition file. It is often convenient to have `NumLock` on when Reflection is run, since in this mode the numeric keypad keys emulate the VT300 series application keypad keys.

The line in the definition file that sets the initial state for CapsLock or NumLock must begin with the word "set," follow the keyboard-ID statement, and precede any keystroke definitions. (These set statements are only valid in a mapping definition file; they are not part of Reflection command language.)

For example, if you are an Enhanced keyboard user and you wish to have `NumLock` forced on and `CapsLock` forced off, you would start your keyboard mapping definition this way:

```
keyboard-ID = enhanced
```

```
set num-lock on
```

```
set caps-lock off
```

Keystroke definitions can follow these statements—they are optional. Compile the mapping definition into a configuration file (as explained on page 43). Initialization of `NumLock` and `CapsLock` occurs whenever that particular configuration file is loaded.

Keystroke Definition: Elements

A keystroke definition equates a keystroke with a task. The left side of the equation is the keystroke, and the right side is the action to be taken in response to that keystroke.

The form of a keystroke definition statement is:

```
<input specification> = <output specification> [;<comment>]
```

Each keystroke definition has these elements:

- ▲ The keystroke being defined, including shift keys
- ▲ An equal sign (=), with a space before and after it
- ▲ The characters, functions, or control codes that should result when the keystroke is performed
- ▲ A comment (optional)

Input specifications are described below, output specifications are covered on page 26, and comments are discussed on page 34.

See page 12 for the keyboard mapping syntax quick reference.

Keystroke Definition: Input Specification

The input specification in keyboard mapping is made up of either a keystroke, or an interrupt 16 or interrupt 9 specification:

```
{ <keyspec> | <int16 spec> | <int9 spec> }
```

Keyboard specifications are explained here; the two interrupt specifications are explained on page 55.

Keystroke specifications include:

```
[<shiftkeys>] <keyname>
```

Shift keys—**Alt**, **Ctrl**, or **Shift**—are optional in keystroke specifications. A key name is always required, and there can be only one key name.

The key name for `[Home]`, for example, is `home`. Using `home` specifies a key name, while `alt home` specifies both a shift key (`alt`) and a key name.

Shift keys are discussed below; key names are discussed on page 21.

Mapping to Default Keystrokes

Reflection provides a file called `DEFAULT.KBM` which shows how keystrokes are used in the default keyboard mapping. Before you plan your keystroke specifications, consult this file—you may want to avoid overwriting existing default mappings. (Look for the file in the directory where Reflection is installed.)

In most cases, if you overwrite a default mapping by mapping a new function to the keystroke, you can map its original function to another keystroke.

Certain keyboard functions, however, cannot be mapped. If the default keystrokes that perform these functions are used for other purposes, the functions will no longer be available from the keyboard. See page 24 for a discussion of these special default keystrokes.

Shift Keys

Shift keys are keys that can be used in conjunction with other keys as modifiers. The shift keys are `[Shift]`, `[Ctrl]`, and `[Alt]`.

A keystroke definition can include one or more shift keys. Refer to the table below for the names used for shift keys in keystroke definitions.

Key	Shift Key Name
<code>[Ctrl]</code>	<code>ctrl</code>
<code>[Alt]</code>	<code>alt</code>
<code>[Shift]</code> (left only)	<code>lshift</code>
<code>[Shift]</code> (right only)	<code>rshift</code>
<code>[Shift]</code>	<code>shift</code>

There are three shift key names for `[Shift]`. You can specify whether you want your mapping to use the left `[Shift]` only, the right `[Shift]` only, or whether you want it to work with either `[Shift]`.

Almost any combination of the shift key names can be used when defining a keystroke. Of course, by using shift keys in various combinations, you increase the number of unique mappings you can create.

The following restrictions apply to shift key names:

- ▲ A particular shift key name cannot be used twice: `alt alt` is not valid.
- ▲ A shift key cannot be used by itself: mapping `ctrl = command-line` is invalid.
- ▲ `Lshift`, `rshift`, and `shift` cannot be used with the alphabetic keys, the cursor keys of the Enhanced keyboard, or certain numeric keypad keys, unless used in combination with `alt` or `ctrl`. Further discussion of these restrictions begins on page 22.
- ▲ A keystroke definition that uses shift keys with keys from the numeric keypad must use the *cursor key name* rather than the numeric key name. See page 22.

Key Names

In a keystroke specification, a key name is always required, and there can be only one key name. The keyboard layouts, pages 49 through 55, show the key names to be used in keystroke specifications.

In most cases, these names are identical to the symbols found on the keys of U.S. versions of the keyboards. In some cases the name varies slightly from the symbol, or describes the function of the key. For example:

- ▲ The name of the key that has both 5 and % is `five`.
- ▲ The name of the space bar (on which no symbol appears) is `space`.
- ▲ The name of the Enter key on the typewriter keypad is `Return`.

The paragraphs that follow discuss how to use certain categories of key names.

Alphabetic Key Names

Most keyboard mapping syntax is not case sensitive—you can make entries to your mapping file in uppercase, lowercase, or any combination of the two. One exception is when you use an alphabetic key name:

- ▲ If you use an alphabetic key name—`a`, `b`, `c`, etc.—in a keystroke definition without shift keys, case sensitivity applies. The following definitions result in two distinct mappings:

```
a = <output spec1> ; type lowercase "a" for <output spec1>
A = <output spec2> ; type uppercase "A" for <output spec2>
```


- ▲ If you use an alphabetic key name in a keystroke definition *with* shift keys, uppercase and lowercase letters are completely interchangeable. The following definitions result in the same mapping:

```
ctrl alt a = <output specification> ; same mapping
ctrl alt A = <output specification> ; same mapping
```

In both cases, you press **Ctrl**-**Alt**-**A** to produce the <output specification>.

The case of the letter “a” works differently in the two examples. In the first example, the uppercase A can be seen as having an implied shift—because pressing **Shift**-**A** is one way to type an uppercase “A.” In the second example, the case of “a” is not important.

When using an alphabetic key name, you cannot use `shift`, `lshift`, or `rshift` as a shift key, except in combination with `alt` or `ctrl`. (See page 20 for a discussion of shift keys.)

Numeric Keypad Key Names

Several keys on the numeric keypad have two possible key names, a numeric key name and a cursor key name. This table shows the numeric key names (“kp” stands for keypad):

kp-1	kp-2	kp-3	kp-4	kp-5	kp-6
kp-7	kp-8	kp-9	kp-0	kp-period	

This table shows the cursor key names:

end	down	pgdn	left	center	right
home	up	pgup	ins	del	

Keyboard layouts, beginning on page 49, show how the two sets of names relate.

With the key names listed in the tables above, you cannot use `shift`, `lshift`, or `rshift` as shift keys, except in combination with `alt` or `ctrl`. This restriction also applies to the `kp-slash` key of the Enhanced keyboard. (See page 20 for a discussion of shift keys.)

When you map a key on the numeric keypad, and the mapping includes a shift key, always use the *cursor key name* to refer to the key—do not use the numeric key name. For example, this is a legal keystroke specification:

```
ctrl shift pgdn = <output specification>
```

and this is not:

```
ctrl shift kp-3 = <output specification> ; illegal use of kp-3
```

When you map a key on the numeric keypad, and the mapping does not include shift keys, you can map to the numeric key name, the cursor key name, or to both.

When you use the cursor key name, you replace the cursor action of the key with your mapping. If you need the cursor action, do not map to the cursor key name—map to the numeric key name, instead, and you will keep the use of the cursor keys.

Similarly, if you need the numeric keys to do data entry map to the cursor key names, not the numeric key names—that way, you keep the use of the numeric keys.

The name that you use will determine the keyboard state required for the mapped keystroke to work—in particular, the states of **NumLock** and **Shift** required to produce the mapped behavior.

When you make normal use of your numeric keypad, you change the states of **NumLock** and **Shift** to control what the keys do. To produce a number, you either toggle **NumLock** on or you press **Shift** with the key. To produce a cursor movement, you toggle **NumLock** off; if **NumLock** is on, you press **Shift** to reverse the effects of **NumLock**.

The same **NumLock** and **Shift** states are required to use mappings made to the numeric or cursor key names. The following table summarizes these requirements:

Numeric Keypad and Required Keyboard States

Keyname Used	NumLock State	Shift Pressed
Numeric	on	no
	off	yes
Cursor	off	no
	on	yes

Cursor Keypad Key Names (Enhanced Keyboard)

The Enhanced keyboard's cursor keypad is located between the alphabetic and numeric keypads. The cursor keypad keys can be mapped separately from the cursor keys on the numeric keypad. The names for the keys follow:

cp-ins	cp-up	cp-home	cp-left	cp-pgup
cp-del	cp-down	cp-end	cp-right	cp-pgdn

When using a cursor keypad key name, you cannot use `shift`, `lshift`, or `rshift` as a shift key, except in combination with `alt` or `ctrl`. (See page 20 for a discussion of shift keys.)

Special Default Keystrokes

Some keystrokes cannot be used in keystroke definitions without eliminating their associated Reflection or terminal emulation functions from the keyboard. These cases are described below.

Menu Shortcut Keys

The ability to open a menu using shortcut keys cannot be mapped to other keystrokes. The menu shortcut keys are:

[Alt]-F	File menu
[Alt]-D	Edit menu
[Alt]-S	Setup menu
[Alt]-K	Keys menu
[Alt]-L	Tools menu
[Alt]-H	Help menu

If you map these keys to perform other keyboard functions, they will no longer work as menu shortcut keys. To reflect this change, the shortcut letters on the menu bar will no longer be highlighted.

However, once you select the menu bar with **[Alt]** or the mouse, the shortcut letters will be highlighted, and you will be able to navigate the menus using the shortcut letters, the **[←]** and **[→]** keys, or the mouse.

Function Keys

The function keys can be mapped to perform non-default functions. However, it is recommended that you do not map them.

If you choose to map `f1`, `f2`, `f3`, etc, the function keys will no longer behave in a context-sensitive manner according to the labels on the screen. Instead, the function keys will always do what you have mapped them to do, regardless of the screen labels.

NumLock and CapsLock

The `NumLock` and `CapsLock` keys can be mapped to perform non-default functions, but the ability to toggle the keyboard's CapsLock or NumLock state cannot be mapped to another keystroke.

F1 through F10

The keystrokes `F1` through `F10` receive special treatment during keyboard mapping.

- ▲ If the labels are visible across the bottom of the screen in Reflection, then `F1` through `F8` perform the action shown on the labels, regardless of any keyboard mapping in effect. Mapping only takes effect if no labels appear.
- ▲ `F9` is mapped to `no-labels` by default—it can be used to blank the labels.
- ▲ `F10` is mapped to `main-menu` by default—it can be used to return to Reflection's main menu labels.

By default no function key labels appear when you run Reflection. This is controlled by the Initial Function Keys option buttons in the Display Setup dialog box.

Keystroke Definition: Output Specification

The output specification defines the characters to be generated and/or the functions to be performed in response to a given keystroke. It can either be `NULL` or a combination of the following:

```
{<keyboard function>|<string>|<number>}
```

`NULL` Specification used to disable a keystroke. Discussion below.

`<keyboard function>`

A name used to map a *keyboard function*. For example, “command-line” is a Reflection function name, and “VT-PF1” is a terminal emulation function name. Keyboard functions that cannot be mapped do not have function names.

See page 27 for a discussion of keyboard functions. Also see page 34 for a table of keyboard function names, and the keystrokes they are mapped to by default.

`<string>`

A *character string* is a series of characters enclosed in quotes; it can include ASCII control codes, which are represented by two-character sequences. See page 28.

`<number>`

A character defined in terms of its *ASCII decimal value*. See page 31.

An output specification can include any combination of up to 127 function names and characters. For example, the following is a valid mapping:

```
alt comma = command-line "set remote-mode no^M^M" home-up
```

The mapping includes a keyboard function, a string of characters in quotes ending with two carriage return control codes, and a second keyboard function. Once this mapping is compiled into a configuration file, pressing **Alt-** brings up the Reflection command line, puts the PC into local mode, does two carriage returns (one to execute the command, and one to exit the command line), and homes the cursor.

NULL

If a keystroke is defined as null, it is completely disabled. For example, **[Alt]-[X]** is the default keystroke for exiting Reflection. If **[Alt]-[X]** is remapped to null, the keystroke has no effect:

```
alt x = NULL
```

Keyboard Functions

Keyboard functions include Reflection keyboard functions and keyboard functions for VT and ANSI terminal emulation. Listed below are page references for tables that show the keyboard functions that can be mapped to non-default keystrokes. The tables give the name used for each of the mappable functions, along with the keystroke to which each function is mapped by default. Keyboard function names of two or more words (such as `clear-display`) must be hyphenated as shown in the tables.

Mappable Keyboard Functions

Reflection functions	page 34
VT terminal functions	page 36
ReGIS graphics functions	page 39
Tektronix graphics functions	page 40
Softkey functions	page 40
ANSI terminal functions	page 40

Two special categories of keyboard functions are discussed below.

Reset Functions

There are some restrictions on how host reset functions (`soft-reset`, `hard-reset`, `communications-reset`, and `exit-to-dos`) are mapped:

- ▲ They can only be mapped to function keys **[F1]** through **[F10]**, or alphabetic keys shifted with **[Ctrl]**, **[Alt]**, or both.
- ▲ They cannot be combined with other functions: they must be the only output specification for a particular keystroke.

Priority Functions

Normally, an output specification for a keystroke is processed in the order that it appears in the mapping file. In the following example, for instance, **[Alt]-T** first shows a local directory listing and then goes to the next session:

```
alt t = command-line "dir^M^M" next-session
```

There are a few keyboard functions, however, that are executed first, no matter where they appear in an output specification:

```
host-break  
print-screen  
exit-to-dos
```

Character Strings

A character string is a series of characters enclosed in quotes, as in the following:

```
"This is a string enclosed in quotes."
```

Most keyboard mapping syntax is not case sensitive—you can make entries to your mapping file in uppercase, lowercase, or any combination of the two. One exception is when you use a character string. Character strings are treated as literals in mapping statements: within the quotes, uppercase and lowercase distinctions are preserved. The character string "This" is different from the character string "this".

Characters that can be entered from the keyboard can appear in a string. Additionally, the ASCII control codes in the range 0–31 can be used in character strings; they will be discussed next. To map ASCII 127 (the delete character), or ASCII characters in the range 128–255 to keystrokes, see page 31.

ASCII Control Codes

A string can contain ASCII control codes in the range 0–31. Within strings, control codes, like carriage return, linefeed, and tab, are represented by two-character sequences; the sequence `^m` or `^M`, for instance, represents a carriage return. See page 29 for a table listing the two-character representations used to include ASCII control codes within character strings. (The table shows the letters in uppercase—you can also type them in lowercase.)

Control Code Representations

ASCII Decimal	Two-character Sequence	Control Code
0	^@	Null
1	^A	Start of header
2	^B	Start of text
3	^C	End of text
4	^D	End of transmission
5	^E	Enquiry
6	^F	Acknowledge
7	^G	Bell
8	^H	Backspace
9	^I	Horizontal tab
10	^J	Linefeed
11	^K	Vertical tab
12	^L	Form feed
13	^M	Carriage return
14	^N	Shift out
15	^O	Shift in
16	^P	Data link escape
17	^Q	Device control 1 (XON)
18	^R	Device control 2
19	^S	Device control 3 (XOFF)
20	^T	Device control 4
21	^U	Negative acknowledge
22	^V	Synchronous idle
23	^W	End of transmission block
24	^X	Cancel
25	^Y	End of medium
26	^Z	Substitute
27	^[	Escape
28	^\ ^\\	Field separator
29	^]	Group separator
30	^^	Record separator
31	^_ ^_	Unit separator

The following keystroke definition means that pressing **[Alt]-[F2]** is the same as typing `SHOWME` and pressing **[Enter ↵]**:

```
alt f2 = "SHOWME ^M"
```

In the following example, the VT control function for setting smooth scrolling is assigned to a keystroke:

```
alt lshift s = command-line "display '^[[?4h' ^M^M"
```

The control function for jump scrolling is assigned to another keystroke:

```
alt lshift j = command-line "display '^[[?4l' ^M^M"
```

The control character for expressing `CSI` (the control sequence introducer) is `^[[` in these examples; it can also be expressed as `^(155)`:

```
alt lshift s = command-line "display '^(155)?4h' ^M^M"
alt lshift j = command-line "display '^(155)]?4l' ^M^M"
```

Special Characters and the Literal Escape Character

The following ASCII characters have special meaning in string definitions:

Character	ASCII Code	Description
Quote (")	34	String delimiter
Single quote (')	39	String delimiter
Caret (^)	94	First character of an ASCII control code
Backslash (\)	92	Default literal escape character

When a backslash, caret, or quote mark is preceded by a backslash—the literal escape character—its ASCII value rather than its special meaning is indicated.

To make the keystroke **[Ctrl]-[4]** display `This is a caret: ^`, enter the following keystroke definition:

```
ctrl four = "This is a caret: \^"
```

Concatenating a String

When you are defining a long string in your keyboard mapping file (.KBM), you may concatenate it so that the entire string appears on the screen as one line. Example:

```
alt f1 = "This string appears on your " &
        "screen as one line. To dis" &
        "play it in YOUR.KBM in this" &
        "format, use ampersands (&).^M"
```

There are a few rules for concatenating strings:

- ▲ No more than three ampersands can be used for one string.
- ▲ A comment line (a semicolon followed by some text) cannot appear on a separate line within the concatenated string. Comments at the end of a line are legal.

ASCII Decimal Values

A character can be defined in terms of its ASCII decimal value. Values in the range 0–127 conform to the ASCII character set. Values in the range 128–255 represent characters in the IBM extended font.

For example, you can use either of the methods below to map the string “Reflection” to the keystroke **[Alt]-[R]**:

```
alt r = "Reflection"
alt r = 82 101 102 108 101 99 116 105 111 110
```

ASCII control codes in the range 0–31 can be specified with a two-character sequence (explained on page 28), or by their ASCII decimal values. To append a carriage return to the second example above, for instance, add the value 13 to the end of the output specification. See page 29 for a listing of ASCII control codes.

When an 8-bit character (ASCII characters 160–255) is mapped to a keystroke, Reflection translates the character from the current PC code page to the host character set when the keystroke is used. You can disable this translation with the command `SET DISABLE-TRANSLATION-KB`; see the *Reflection Command Language Manual*.

Reflection Commands

Reflection command language (RCL) commands and files can be mapped to keystrokes using the `command-line` keyboard function. This keyboard function is equivalent to displaying the Reflection command line.

In your definition, follow `command-line` with the text that you would normally type in the Reflection command line—a Reflection command or the name of a command language file—enclosed in quotes. Within the quotes, use `^M` to include the carriage return character in your mapping. Typically, two carriage return characters should follow your text:

- ▲ The first carriage return executes the RCL command, or RCL file
- ▲ The second carriage return closes the command line when execution is complete

This example maps `[Ctrl]-[Shift]-[C]` to an RCL command to load a configuration file called `HOST2.CFG`:

```
ctrl shift c = command-line "load host2.cfg^M^M"
```

In this example, `[Alt]-[D]` executes the RCL file called `DOWNLOAD.RCL`, which resides in the `\REFLECT` directory:

```
alt d = command-line "c:\\reflect\\download.rcl^M^M"
```

The literal escape character `\` is used in the sequence `\\` in order to include the backslash character—a special character—in the DOS path specification. See page 30 for a discussion of special characters and the literal escape character.

VT Control Functions

VT terminal control functions can be mapped to keystrokes using the `command-line` keyboard function, and the Reflection command language `DISPLAY` command.

Read the preceding discussion of Reflection command language mappings. Use the RCL `DISPLAY` command to “display” an escape sequence or control function to the terminal—Reflection will respond as if the escape sequence or control function came from a host computer.

The following example maps **Alt-1** and **Alt-2** to display VT control functions that change video character attributes (VT control functions require VT emulation—use term = VT):

```
alt one = command-line "display '^[[7m'^M^M" ; selects inverse video
                                           ; character attribute
alt two = command-line "display '^[[0m'^M^M" ; turns off video
                                           ; attribute
```

In the definitions above, double quotes are used to delimit the string of characters that would normally be typed on the command line, the outer string. Single quotes are used as delimiters for the RCL command, the inner string: `display '^[[7m'`.

The rules for using quotes follow:

- ▲ The same type of quote must be used to mark the beginning and end of a string
- ▲ If one type of quote is used to mark an outer string, the other type can be used for the inner string

In the preceding definitions, the pairs of quotes could have been reversed: the single quotes could have marked the outer string, and the double quotes, the inner string.

The control sequence introducer (^CSI) control character is represented by `^[[`. The table that follows shows how other commonly used VT control characters can be represented in mapping statements:

VT Control Character Representations

Name	Character	Representation
Escape	ESC	^[[
Device Control String	DCS	^[P
Control Sequence	CSI	^[[

Keystroke Definition: Comment

A keystroke definition can include an optional comment. As shown in the example below, comments are delimited by semicolons (;), and can appear at the end of a definition, or on a separate line:

```
alt x = null      ; disables exit
alt cp-right = command-line "display '^[[?5h' ^M^M"
; uses VT control function to set screen display to inverse video
```

The KEYCOMP utility (which compiles your keyboard mapping file into a configuration file) ignores everything between the semicolon and the end of the line.

There is one exception to these general rules: if you are concatenating lines in a keystroke definition, you cannot place comments on separate lines between the concatenated lines. Comments at the end of a line are legal. See page 31 for a discussion of concatenated keystroke definitions.

Reflection's Default Keyboard Mapping

The tables that follow show the Reflection, VT terminal, and ANSI terminal keyboard functions that can be mapped to new keystrokes, along with the keystrokes they are mapped to by default. See page 24 for a discussion of special default keystrokes that perform keyboard functions that cannot be mapped to new keystrokes.

The first column of the table identifies the function. The second column shows the keystroke definition as it applies to all keyboards. Exceptions (keystroke definitions that differ based on the keyboard used or some other special requirement) are listed in the third column.

Reflection Keyboard Functions

Beginning of line	home = home-left	
CI mode	ctrl f8 = ci-mode	
Clear display	alt j = clear-display	
Command line	alt f10 = command-line	
Connection reset	ctrl f8 = connection-reset	
Datacomm statistics	alt f5 = error-recap	
Delete char	del = delete-char	
Disconnect	ctrl f10 = disconnect	
End-of-line	end = home-right	
Exit to DOS	alt x = exit-to-dos	
File transfer screen	<key> = file-xfer-screen	No default key
Help menu	alt h = help-screen	
Home down	ctrl cp-end = home-down ctrl end = home-down	Enhanced
Home up	ctrl cp-home = home-up ctrl home = home-up	Enhanced
Insert char	ins = insert-char	
Main menu labels	alt a = main-menu f10 = main-menu	
Modem dialer	ctrl f7 = modem-dialer	
Next page	ctrl cp-pgdn = next-page ctrl pgdn = next-page	Enhanced
Next session	alt n = next-session	
No labels	f9 = no-labels	
PF key labels	<key> = pf-key-labels	No default key
Prev page	ctrl cp-pgup = prev-page ctrl pgup = prev-page	Enhanced
Reset video	alt equal = reset-video	
Scroll down	ctrl cp-dn = scroll-down ctrl down = scroll-down	Enhanced
Scroll left	ctrl cp-left = scroll-left ctrl left = scroll-left	Enhanced
Scroll right	ctrl cp-right = scroll-right ctrl right = scroll-right	Enhanced
Scroll up	ctrl cp-up = scroll-up ctrl up = scroll-up	Enhanced
Setup menu	alt s = set-up-keys	
State save	alt b = exit-with-save	

VT Terminal Keyboard Functions*VT Keypad*

Find	cp-ins = find alt home = find	Enhanced
Insert here	cp-home = insert-here alt up = insert-here	Enhanced
Move down	cp-down = move-down down = move-down	Enhanced
Move left	cp-left = move-left left = move-left	Enhanced
Move right	cp-right = move-right right = move-right	Enhanced
Move up	cp-up = move-up up = move-up	Enhanced
Next screen	cp-pgdn = next-screen alt right = next-screen	Enhanced
Previous screen	cp-end = prev-screen alt center = prev-screen	Enhanced
Remove	cp-pgup = remove alt pgup = remove	Enhanced
Select	cp-del = select alt left = select	Enhanced

VT Function Keys

VT F1	alt one = vt-f1
VT F2	alt two = vt-f2
VT F3	alt three = vt-f3
VT F4	alt four = vt-f4
VT F5	alt five = vt-f5
VT F6	alt six = vt-f6
VT F7	alt seven = vt-f7
VT F8	alt eight = vt-f8
VT F9	alt nine = vt-f9
VT F10	alt zero = vt-f10
VT F11	alt q = vt-f11
VT F12	alt w = vt-f12
VT F13	alt e = vt-f13

VT F14	alt r = vt-f14
VT F15 (Help)	alt t = vt-f15
VT F16 (Do)	alt y = vt-f16
VT F17	alt u = vt-f17
VT F18	alt i = vt-f18
VT F19	alt o = vt-f19
VT F20	alt p = vt-f20

VT Editing Keypad

VT 0	kp-0 = vt-0
VT 1	kp-1 = vt-1
VT 2	kp-2 = vt-2
VT 3	kp-3 = vt-3
VT 4	kp-4 = vt-4
VT 5	kp-5 = vt-5
VT 6	kp-6 = vt-6
VT 7	kp-7 = vt-7
VT 8	kp-8 = vt-8
VT 9	kp-9 = vt-9
VT period	kp-period = vt-period
VT enter	kp-enter = vt-enter
	kp-plus = vt-enter
VT comma	kp-plus = vt-comma
	kp-minus = vt-comma
	kp-star = vt-comma
VT minus	kp-minus = vt-minus
	kp-star = vt-minus
	kp-minus = vt-minus

Enhanced

Enhanced
AT keyboard
PC keyboard
Enhanced
AT keyboard
PC keyboard

F1 through **F8***

VT PF1 (GOLD)	f1 = vt-pf1
VT PF2	f2 = vt-pf2
VT PF3	f3 = vt-pf3
VT PF4	f4 = vt-pf4
Move up	f5 = move-up
Move down	f6 = move-down
Move left	f7 = move-left
Move right	f8 = move-right

VT User-Defined Keys

UDK6	shift alt six = udk6
UDK7	shift alt seven = udk7
UDK8	shift alt eight = udk8
UDK9	shift alt nine = udk9
UDK10	shift alt zero = udk10
UDK11	shift alt q = udk11
UDK12	shift alt w = udk12
UDK13	shift alt e = udk13
UDK14	shift alt r = udk14
UDK15	shift alt t = udk15
UDK16	shift alt y = udk16
UDK17	shift alt u = udk17
UDK18	shift alt i = udk18
UDK19	shift alt o = udk19
UDK20	shift alt p = udk20

* When no function key labels are displayed, or the PF-keys are displayed, F1 through F8 are mapped to VT functions. Otherwise these keys perform the action on the corresponding label.

Other VT Operations

Backarrow	backspace = backarrow-key	
Backarrow opposite	ctrl backspace = not-backarrow-key	
Communications reset	ctrl f1 = communications-reset	
Compose key	alt f8 = compose-key	
Disconnect	ctrl f10 = disconnect	
Hard reset	ctrl f3 = hard-reset	
Hold screen	scroll-lock = hold-screen	
	alt scroll-lock = hold-screen	
	shift scroll-lock = hold-screen	
Hold screen set	ctrl s = hold-screen-set	
Hold screen clear	ctrl q = hold-screen-clear	
Host break	ctrl pause = host-break	Enhanced
	ctrl scroll-lock = host-break	
Paste clipboard	ctrl v = paste-clipboard	
Print screen	alt print-screen = print-screen	Enhanced
	alt kp-star = print-screen	
Return	return = return	
Send answerback	alt f7 = send-answerback	
Soft reset	ctrl f2 = soft-reset	
Start select mode	alt c = select-mode	
Toggle print logging	ctrl print-screen = toggle-print-logging	Enhanced
	ctrl kp-star = toggle-print-logging	

ReGIS Graphics Keyboard Functions

Toggle graphics/text	alt v = toggle-graphics
Press button 1	button-1-down = vt-button-1-down
Release button 1	button-1-up = vt-button-1-up
Press button 2	button-2-down = vt-button-2-down
Release button 2	button-2-up = vt-button-2-up
Press button 3	button-3-down = vt-button-3-down
	alt button-1-down = vt-button-3-down
Release button 3	button-3-up = vt-button-3-up
	alt button-1-up = vt-button-3-up
Press button 4	alt button-2-down = vt-button-4-down
Release button 4	alt button-2-up = vt-button-4-up

Tektronix Graphics Keyboard Functions

Cancel bypass	alt p = tek-page
	alt r = tek-page
Enter/Exit Tektronix	alt g = toggle-tek
Erase display	alt p = tek-page
Local/Online toggle	alt l = tek-toggle-online
Reset	alt r = tek-reset
Scaled/Unscaled toggle	alt z = tek-zoom
Set alpha mode	alt p = tek-page

Softkeys Graphics Keyboard Functions

Softkey 1*	shift f1 = s1
Softkey 2	shift f2 = s2
Softkey 3	shift f3 = s3
Softkey 4	shift f4 = s4
Softkey 5	shift f5 = s5
Softkey 6	shift f6 = s6
Softkey 7	shift f7 = s7
Softkey 8	shift f8 = s8
Softkey labels	alt f9 = soft-keys

ANSI Terminal Keyboard Functions

F1 through **F12**†

F1	f1 = ansi-f1
F2	f2 = ansi-f2
F3	f3 = ansi-f3
F4	f4 = ansi-f4
F5	f5 = ansi-f5
F6	f6 = ansi-f6
F7	f7 = ansi-f7
F8	f8 = ansi-f8

* Use F1 through F8 when the softkeys are visible, otherwise use the following softkeys.

† When a set of function key labels is displayed, F1 through F12 perform whatever action is indicated on the corresponding label; when none are shown, these keys perform ANSI functions. When Reflection labels are up, press F9 to make them go away. To get Reflection labels back again, press ALT-A.

F9	f9 = ansi-f9	
F10	f10 = ansi-f10	
F11	f11 = ansi-f11	Enhanced
	alt q = ansi-f11	
F12	f12 = ansi-f12	Enhanced
	alt w = ansi-f12	

Keypad Keys

Home	home = ansi-home	
Home	cp-home = ansi-home	Enhanced
Page up	pgup = ansi-pgup	
	cp-pgup = ansi-pgup	Enhanced
Center (no function)	center = ansi-center	
End	end = ansi-end	
	cp-end = ansi-end	Enhanced
Insert character	ins = ansi-ins	
	cp-ins = ansi-ins	Enhanced
Delete character	del = ansi-del	
	cp-del = ansi-del	Enhanced
Page down	pgdn = ansi-pgdn	
	cp-pgdn = ansi-pgdn	Enhanced

Compiling the Mapping

A keyboard mapping file must be compiled into a Reflection configuration file.

First, decide on a configuration file in which to store the keyboard information. R2.CFG and R4.CFG are reserved as Reflection's default configuration filenames. Since the default configuration file is loaded automatically when no other configuration file is named, you may want to use another name initially.

KEYCOMP Syntax

KEYCOMP is a stand-alone program that compiles keyboard definition statements into a format that Reflection can interpret. It then inserts this information into the designated configuration file. The program is run at the DOS prompt.

The syntax is as follows:

```
KEYCOMP <keyboard mapping filename> <configuration filename>
```

For example:

```
C:> KEYCOMP MYKEYS.KBM MYNEW.CFG
```

For information on KEYCOMP error messages, see pages 44–47 in this manual.

Once you have successfully compiled your keyboard mapping file, run Reflection using this configuration file. For example, to use MYNEW.CFG, type one of the following at the DOS prompt:

```
C:> R2 MYNEW.CFG
```

```
C:> R4 MYNEW.CFG
```

All keys not specifically mapped retain their default Reflection values.

Returning to Default Configuration Values

You can return a configuration file to default values for all configuration values, your keyboard mapping only, or all configuration values, except keyboard mapping.

Complete Default Configuration

If you used R2.CFG or R4.CFG as your configuration filename and want to reset all of Reflection's original values (including keyboard mapping defaults), reload Reflection with the /D switch. Alternatively, you can delete R2.CFG (or R4.CFG) altogether and then reload Reflection. Resave your new configuration under the default filename.

Keyboard Mapping Default Configuration

Make a keyboard mapping file with keyboard identification and terminal type entries. Compile the mapping file into your configuration file. The resulting configuration file will have default keyboard mapping settings, with all of your other configuration values intact.

Default Configuration, Except Keyboard Mapping

If you run KEYCOMP with a keyboard mapping file and the name of a configuration file that does not yet exist, KEYCOMP creates a configuration file that contains your mapping information, and all other configuration values will return to their defaults.

KEYCOMP Error Messages

Caps Lock state previously defined

The initial state of Caps Lock can only be defined once in a keyboard mapping file.

ERROR: Continuation not found

An ampersand (&) was found in the keyboard mapping file, but the expected continuation line did not follow it.

ERROR: Continuation not last token of line

An ampersand (&) is a reserved character that allows you to concatenate a long string in your keyboard mapping file. It must be the last character of a line.

ERROR: Equals sign (=) expected

Each keyboard mapping statement requires an equals sign.

ERROR: Invalid combination of shift keys

Shift keys (Shift, Ctrl, and Alt) cannot be used twice, and must be combined with another key or keys.

ERROR: Invalid control code

ASCII control codes in the range 0–31 can be specified. The first character of an ASCII control code is a caret (ASCII code 94).

ERROR: Invalid keyboard ID

The keyboard ID you specified in the keyboard statement of your keyboard mapping file (for example, TEST.KBM) cannot be used with this Reflection product. This message also occurs if `keyboard-id =` has no keyboard ID assigned.

ERROR: NULL must be only outspec

The NULL keystroke must be the only specification for a key: it cannot be combined with another specification.

ERROR: Open string

An opening quote is included in a keystroke definition, but there is no closing quote.

ERROR: Outspec may not exceed 127 characters/functions

Up to 127 characters or functions can be assigned to a single keystroke in a keyboard mapping file. Your specification has exceeded this limit.

ERROR: Reset function must be only outspec

Reset functions (`hard-reset`, `soft-reset`, `communications-reset`, or `exit-to-dos`) cannot be combined with other functions: they must be the only output specification for a particular keystroke.

ERROR: Syntax error

Type `KEYCOMP /?` to see the correct syntax.

ERROR: Table overflow—too many remapped keystrokes

Depending on the size of the strings in a keystroke definition, up to 512 keystrokes can be mapped.

ERROR: Too many continuations

A maximum of 3 continuations, represented by an ampersand (&) at the end of a line, can be used to concatenate a long string assigned to one keystroke.

ERROR: Value is out of range

You can use a decimal representation for a character; the value must be in the range 0–255. Also see the description of “ERROR: Invalid control code.”

Expected “TERM” constant

No terminal specification statement was included in the keyboard mapping file. If you are using Reflection’s ANSI emulation, this file must include `term = ansi`. A terminal type of VT is otherwise assumed.

Format: KEYCOMP <source file> <config file>

You must specify a source file (such as TEST.KBM) and a configuration file (such as R4.CFG) in order to compile mapping statements into a configuration file.

Invalid Set statement

Set statements define the initial state of NumLock and CapsLock and must precede keyboard mapping statements for keystroke definitions.

Invalid term—must be “VT” or “ANSI”

If a keyboard mapping file (such as TEST.KBM) does not indicate a term type, then VT is assumed.

Keyboard ID not specified

A keyboard mapping file (such as TEST.KBM) must include a statement that specifies the keyboard-ID, for example: `keyboard-id = enhanced`.

KEYCOMP: Config file version not compatible with keyboard mapping

A configuration file created with a version of Reflection 2.2 or earlier cannot be updated with this version of the keyboard mapping program.

KEYCOMP: Error opening config file

The configuration file (such as R4.CFG) could not be found in the directory specified, or you do not have access to this file.

KEYCOMP: Error opening source file

The source file (such as TEST.KBM) could not be found in the directory specified.

KEYCOMP: Error reading source file

The source file (such as TEST.KBM) was found, but the format is incorrect or there is some other problem reading the file.

KEYCOMP: Error writing to config file

This message typically appears if the disk space available is not sufficient for storing the configuration file.

KEYCOMP: Output file is not a compatible Reflection configuration file

The configuration file you specified (such as R2.CFG) is not in the correct format.

Num Lock state previously defined

The initial state of Num Lock can only be defined once in a keyboard mapping file.

Reflection configuration file was not created/updated

This typically appears with another KEYCOMP error message indicating the specific problem encountered when you ran KEYCOMP.

Keyboard Layouts

This chapter shows the layout for each of the valid keyboard-IDs:

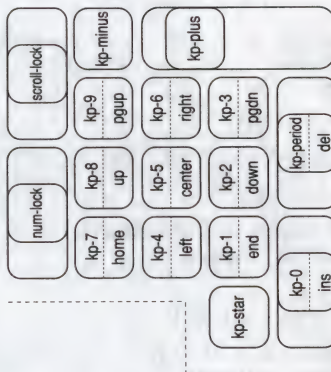
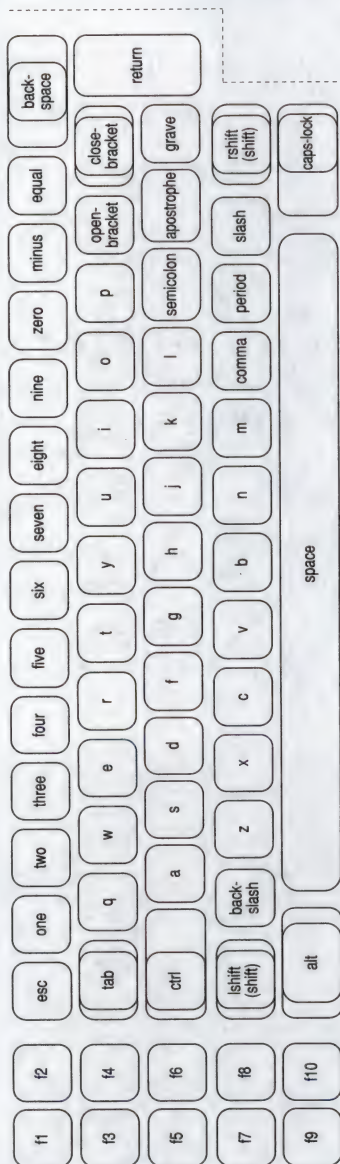
- IBM PC or XT
- IBM PC/AT
- IBM Enhanced 101/102
- Digital LK250
- HP Vectra (non-Enhanced)
- Key Tronic 5151

These layouts will help you specify keystrokes in your mapping file: find the layout that corresponds to your keyboard and then use the key names *as they are shown*. For instance, the keystroke **Alt-1** is specified in a mapping file as `alt one`, *not* as `alt 1`:

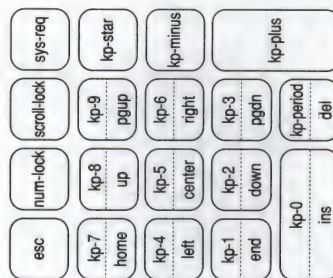
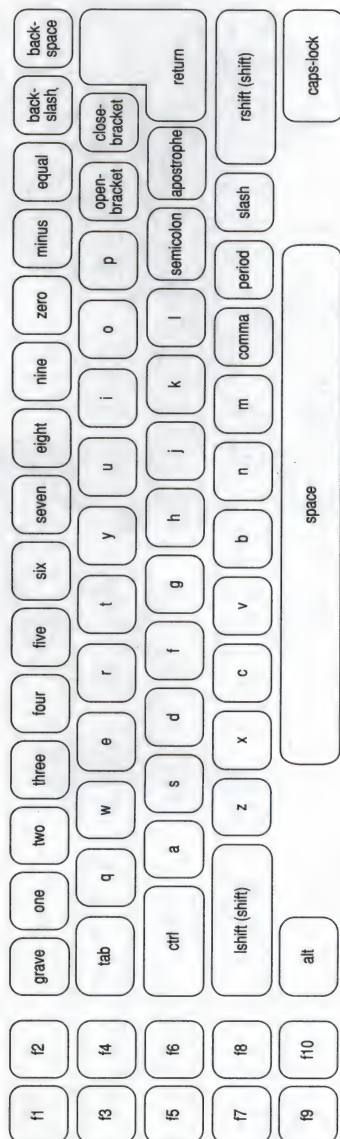
```
alt one = "allin1^M"
```

Keyboard mapping syntax is generally *not* case sensitive—you can type key names in uppercase, lowercase, or any combination of the two. The exception to this rule is discussed on page 21.

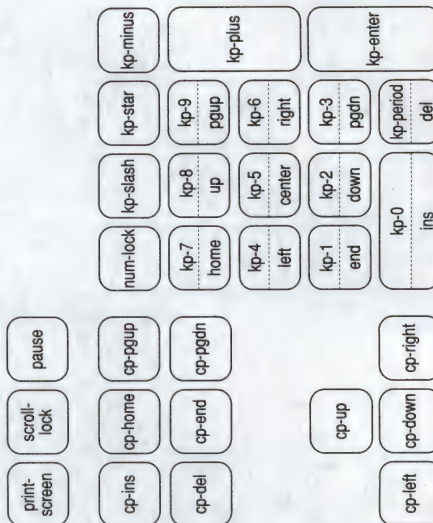
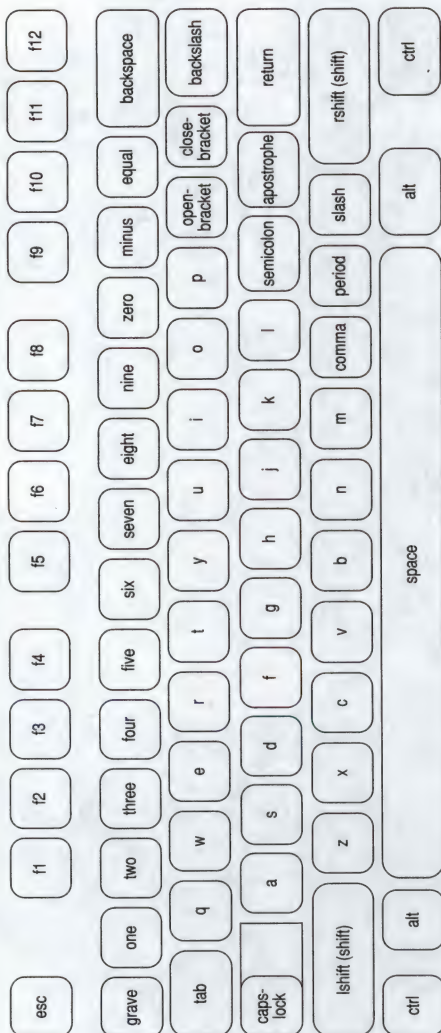
IBM PC or XT Key Names



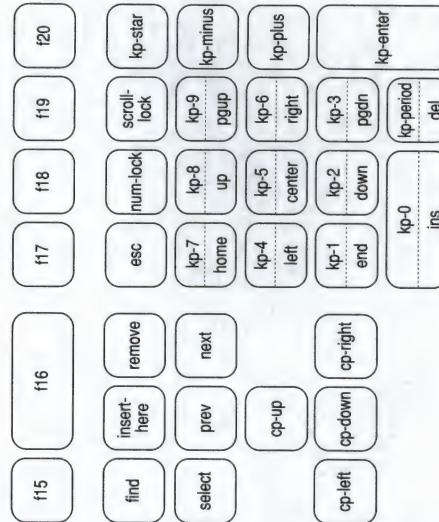
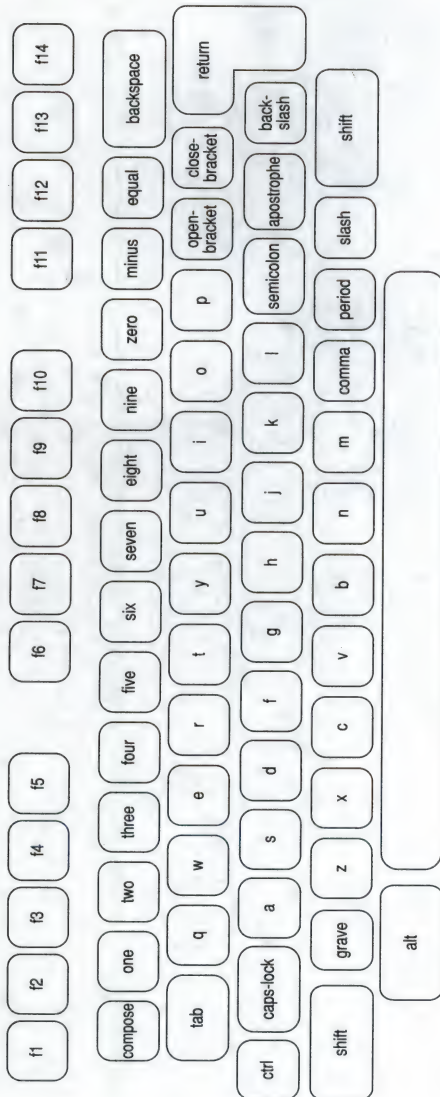
IBM PC/AT Key Names



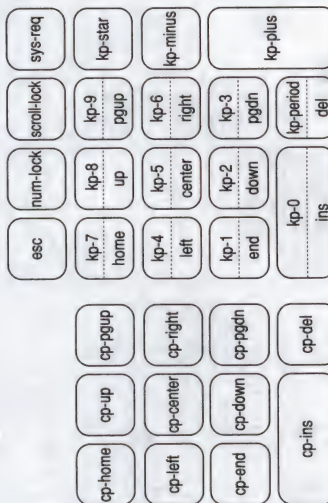
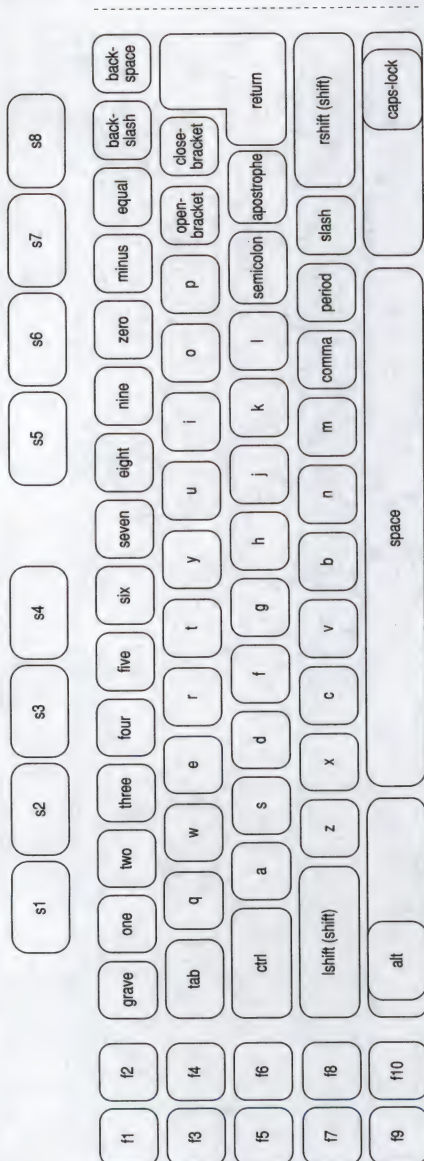
IBM Enhanced 101/102 Key Names



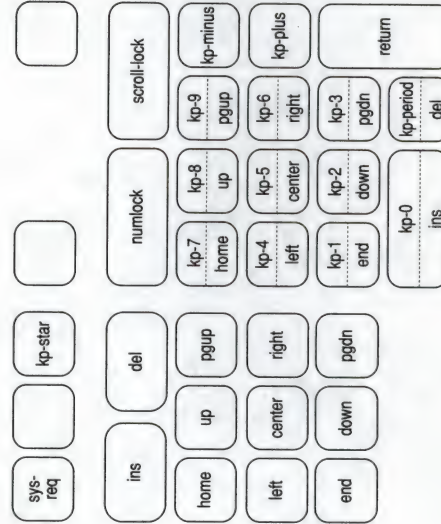
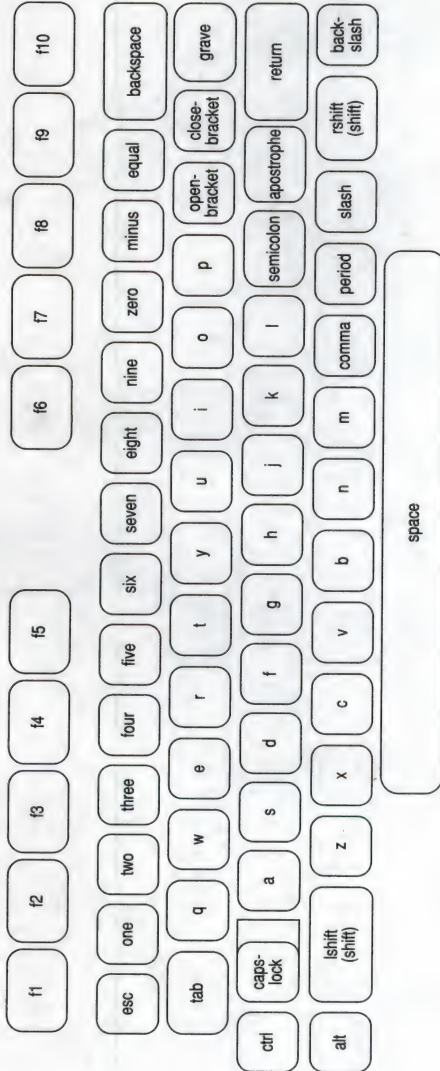
Digital LK250 Key Names



HP Vectra (non-Enhanced) Key Names



Key Tronic 5151 Key Names



Other Methods of Defining Keystrokes

The standard method of defining a keystroke is by means of a key name. A keystroke can also be described in terms of its appearance at software interface levels interrupt 16 and interrupt 9. The material in this section assumes that you are familiar with these interface levels.

These additional methods are included to allow limited support of non-standard keyboards. If your keyboard is supported by the KEYCOMP program, you should not have to concern yourself with doing keystroke mappings via interrupt 16 or interrupt 9 specifications; specifying the key name is sufficient. See page 13 for a list of supported keyboards.

Keyboard Diagnostics

A program called KEYMON is distributed with Reflection to help you determine how your keyboard and keyboard BIOS interpret keystrokes. KEYMON is a tool for diagnosing keyboard problems. Keystrokes should normally be defined by name, as indicated on the keyboard diagram that corresponds to your keyboard type.

Type `KEYMON` at the DOS prompt; exit the program by pressing `[E]`. The complete output on your screen looks something like this:

```
-----  
KEYMON Version 4.30 -- Keyboard Diagnostic Program  
Copyright (C) 1987-92 Walker Richer & Quinn, Inc.  
Type "e" to exit.  
-----
```

```
PC keyboard BIOS is Enhanced  
Hardware type is FCh - IBM PC/AT  
Interrupt 15 Function 4F is available
```

```
Interrupt 9 scan code:12h, 18d  
Interrupt 16 returns: AH=12h, 18d   AL=65h,101d  <e>  
Interrupt 9 scan code:92h,146d
```


The output provides the following information:

- ▲ Whether the PC keyboard BIOS is standard or Enhanced
- ▲ What the value of the hardware flag is (*FCh* indicates an AT, *FF* is a PC, *FE* is an XT, *F8* is a PS/2 Model 80, and *FA* is a PS/2 Model 30)
- ▲ Whether support for the interrupt 15 Function 4F for hooking keyboard interrupts is available

Determining Interrupt 9 and 16 Values

KEYMON monitors both the hardware and software keyboard interrupts, reporting the scan code of any keyboard interrupts it sees and any characters that become available at interrupt 16 (generated by the BIOS in response to keyboard interrupts).

By running KEYMON and pressing keys, you can see both how the keyboard is acting and how the BIOS is reacting. For example, here's what KEYMON reports if you press **M** (unshifted) on your keyboard:

```
Interrupt 9 scan code: 32h, 50d
Interrupt 16 returns: AH=32h, 50d  AL=6Dh,109d  <m>
Interrupt 9 scan code: B2h,178d
```

The first line reports the keyboard interrupt that occurred when **M** was pressed; its scan code is 50. The second line reports the output that became available via interrupt 16 because the BIOS generated a character in response to seeing the **M** key pressed. It consists of a high byte (AH), which in this case is the key's scan code, and a low byte (AL) that contains the ASCII code for the character "m." The third line reports the scan code generated when the key was released.

When you're through retrieving scan codes, press **E** to exit KEYMON.

Using Interrupt 9

The keystroke specification in a keystroke definition can be given in terms of the way the keystroke appears at the hardware interrupt level. The syntax for such an input specification is as follows:

```
[<shiftkeys>] <number> [HIDE] [E0-PREFIXED] [E1-CTRL-PREFIXED]
```

<Shiftkeys>

The syntax for <shiftkeys> is:

```
{SHIFT|ALT|CTRL|LSHIFT|RSHIFT|IGNORE-SHIFTS}
```

When you specify `IGNORE-SHIFTS`, Reflection ignores whether a left shift or right shift is used. In the example below, `sys-req` is mapped to `Tab` no matter how it is shifted. (The text after the semicolon is a comment.)

```
IGNORE-SHIFTS 84 = tab-key ;maps 84 (sys-req) to Tab
```

The `IGNORE-SHIFTS` parameter is valid only when used with an interrupt 9 specification. See “Shift Keys” on page 20 for a description of the other shift options.

<Number>

<Number> is the decimal number returned by KEYMON, in the range 0–255.

When you press `PgUp` on the keypad, for instance, KEYMON returns something similar to the following:

```
Interrupt 9 scan code:49h, 73d
Interrupt 16 returns: AH=49h, 73d   AL=00h, 0d < >
Interrupt 9 scan code:C9h,201d
```

The hex value of the key is 49 (shown as `49h` by KEYMON), and its decimal value is 73 (shown as `73d`). A keystroke specification must be given in terms of its decimal value. To map the VT F6 function to `PgUp`, use the following:

```
73 = vt-f6 ;maps 73 (kp-pgup) to VT F6
```


HIDE

The interrupt 9 HIDE option indicates that the keystroke should not be passed on to any other routines in the keyboard interrupt chain. For instance, you can map the **M** key at the interrupt 9 interface level to the Reflection function `user-keys`. The definition would look like this in your keyboard mapping file:

```
50 = user-keys
```

Each time **M** (unshifted) is pressed, the user-key labels are displayed and, if you are in remote mode, an `m` is transmitted. The **M** is interpreted at the interrupt 9 level as the `user-keys` function and at the interrupt 16 level as the character `M`, thus producing two unrelated results.

To restrict the definition of **M** to the Reflection `user-keys` function, use the following definition:

```
50 HIDE = user-keys
```

Prefix Options

The interrupt 9 `E0-PREFIXED` and `E1-CTRL-PREFIXED` options refer to prefix scan codes issued by Enhanced 101/102 type keyboards that use an Enhanced BIOS. They allow you to distinguish between duplicate keys on the key and cursor pads.

Using Interrupt 16

A keystroke specification can be a keystroke as it appears at the interrupt 16 interface level. The syntax for such a keystroke definition is as follows:

```
[<shiftkeys>] <number> <number>
```

<Shiftkeys>

The `<shiftkeys>` parameter is discussed on page 20.

<Number>

The <number> parameters represent the decimal AH and AL values returned by KEYMON for interrupt 16. Each number can be in the range 0–255.

To make an unshifted **[Alt]-[M]** display the system keys, first get the AH and AL values for **[Alt]-[M]**. Run KEYMON and press the keystroke combination. You should see output similar to the following:

```
Interrupt 9 scan code:38h, 56d
Interrupt 9 scan code:32h, 50d
Interrupt 16 returns: AH=32h, 50d   AL=00h, 0d < >
Interrupt 9 scan code:B2h,178d
Interrupt 9 scan code:B8h,184d
```

The keystroke definition then looks like this:

```
alt 50 0 = system-keys
```

To make (right) **[Shift]-[F10]** home the cursor, run KEYMON and press that particular keystroke. You should see something similar to this:

```
Interrupt 9 scan code:36h, 54d
Interrupt 9 scan code:44h, 68d
Interrupt 16 returns: AH=5Dh, 93d   AL=00h, 0d < >
Interrupt 9 scan code:B6h,182d
Interrupt 9 scan code:C4h,196d
```

To map the Reflection home-up function to (right) **[Shift]-[F10]**, use the following keystroke definition:

```
rshift 93 0 = home-up
```

See the *IBM Technical Reference* manual for a description of the interrupt 9 and interrupt 16 keyboard interfaces. In general, use interrupt 16 if you can. This way BIOS replacement routines, like the ones in the IBM supplied keyboard drivers, still function.

VT340 Graphics

Reflection 4 supports Digital's *Remote Graphics Instruction Set*, as implemented on Digital's VT330/340 series. ReGIS is a symbol system that describes the parts of an image using standard geometric forms: lines, dots, arcs, circles, and curves. You can also use ReGIS to create fill patterns and text. The graphics you create with ReGIS can be used for display or printing.

Chapter 6, *Using ReGIS Graphics*, provides an overview of Reflection's ReGIS features and a sample graphics session.

Chapter 7, *Introduction to ReGIS Commands*, describes the conventions, syntax, and usage of the 10 basic ReGIS commands.

Chapters 8–17 provide detailed information on how to use each of the ReGIS commands.

Chapter 6

Using ReGIS Graphics	69
Graphics Mode	69
Sample Session	70
Mouse Support	71

Chapter 7

Introduction to ReGIS Commands	73
Conventions	73
Command Syntax	73
Command Key Letter	74
ReGIS Command Summary	74
Parentheses	75
Brackets	75
Quotes	76
Control Characters	77
Resynchronization	77
Pixel Vector Directions	78

Chapter 8

Position Command	79
Cursor Positioning with [X,Y]	79
Cursor Positioning with Pixel Vectors	80
PV Multiplication	80
Begin Bounded Position Stack	81
Start Unbounded Position Stack	81
Null Position Argument	82

Chapter 9

Vector Command	83
Draw Dot	83
Draw Line	83
Draw Lines Using Pixel Vectors	84
Begin Bounded Position Stack	84
Start Unbounded Position Stack	85
Temporary Write Control	85

Chapter 10

Curve Command	87
Center of Circle at Current Position	87
Center of Circle at Specified Position	88
Arc with Center at Current Position	88
Arc with Center at Specified Position	89
Closed Curve Sequence	90
Open Curve Sequence	91
Temporary Write Control	92

Chapter 11

Screen Command	93
Display Addressing	93
Scrolling	93
Print Screen	94
Output Mapping Values	95
Monochrome Graphics	96
Color Graphics	96
Background Intensity	100
Time Delay	100
Screen Erase	100
Graphics Cursor	101
Output Cursor Control	101
Input Cursor Control	101

Chapter 12

Write Command	103
Select Standard Pattern	104
Select Binary Pattern	104
Pixel Vector Multiplication	105
Foreground Intensity	106
Writing Styles	107
Overlay Writing	107
Replace Writing	108
Complement Writing	109
Erase Writing	110
Negative Pattern Control	110

Shading	110
Examples of Writing Styles	112
Plane Selection	115

Chapter 13

Polygon Fill Command	117
Vector	117
Curve	118
Position with Curve	118
Temporary Write Control	119

Chapter 14

Text Command	121
About Character Formats	121
About Text Formats	122
Select Character Set	123
Character Spacing	125
Character Size	126
Height Multiplier	127
Size Multiplier	128
String Tilt	128
String and Character Tilt	129
Italics	130
Temporary Text Control	131
Temporary Write Control	131
Positioning Text with PV Values	132

Chapter 15

Load Command	133
Select Character Set	133
Specify Name	133
Load Character Cell	134

Chapter 16

Report Command	137
Cursor Position	137
Macrograph Contents and Storage	137
Character Set	138
Error	138
Graphics Input Modes	139
Interactive Position of Input Cursor	140
Cursor Movement	140
Position Report Format	141

Chapter 17

Macrograph Command	143
Define Macrograph	143
Invoke Macrograph	143
Clear Single Macrograph	144
Clear All Macrographs	144

Using ReGIS Graphics

Reflection 4 allows the IBM PC or compatible to emulate the Digital VT241 and Tektronix 4014 terminals. It also supports ReGIS graphics as implemented on Digital's VT330/340 series. The following features are included:

- ▲ 16 colors
- ▲ Polygon fill
- ▲ Shading with selected patterns
- ▲ Rubberized cursors
- ▲ Double-width and double-height characters
- ▲ Rotated and italicized character sets
- ▲ Mouse support
- ▲ A scaled image showing the complete ReGIS screen (800×480 pixels) on the physical display

All of the emulation and communications features of Reflection 2—file transfer, command language, keyboard mapping, downline loadable font support, and multitasking—are included in Reflection 4.

To run Reflection 4, you need R4.EXE and DECFONT.FT. (You can load Reflection 4 without DECFONT.FT, but you will not be able to display an unscaled image, and only the default resolution will be possible.)

Graphics Mode

Graphics is a mode in which the EGA or VGA card is placed. Features such as double-width, double-height characters and smooth scrolling are supported in graphics mode. Most PC applications run in text mode, which is faster. If your application switches from one mode to the other, your graphics image is not preserved unless you have # Bit Planes Saved set to 4 in the ReGIS Setup dialog box.

Press **Alt-V** to toggle between text mode and graphics mode.

Sample Session

The following instructions walk you through loading Reflection 4 and getting into ReGIS. To create a short ReGIS program with a text editor and then run it in Reflection:

1. Load Reflection 4 by typing `R4` at the DOS prompt. Make sure that the file `DECFONT.FT` is available if you want your resolution to be higher than the 640×350 default. Resolution Choice is configured in the Video Setup dialog box. See "Video Setup" in the *Technical Reference*.
2. Choose General from the Setup menu to open the General Setup dialog box.
3. Clear the check box for Online. This puts Reflection in local mode.
4. Choose OK to activate the new setup.
5. Toggle to DOS with `(Alt)-(right)(Shift)`.
6. To create a file with ReGIS commands, run a text editor and create a file called `BOX.DAT`.
7. Type the following lines:

```
DISPLAY "^[P3p"
DISPLAY "P[100,100]"
DISPLAY "V[+200] [, +100] [-200] [, -100]"
DISPLAY "^[\\"
```

The first line of the file is the escape sequence that puts Reflection into ReGIS mode at the start of a new ReGIS command, `ESC P3p` (the two characters `^[` represent the `ESC` character that starts the sequence). See the *Programmer Reference* card in the *Technical Reference* for a list of other ReGIS escape sequences. The final command instructs Reflection to exit ReGIS mode.

When the file is run, these commands (which are displayed at the bottom of the screen) position the cursor and draw a rectangle.

8. Save the file. The text editor can be left in memory so that you can toggle to it from Reflection.
9. Toggle to Reflection and display the command line. (Choose Command Line from the Tools menu or press `(Alt)-(F10)`.)

10. Type `TYPE BOX.DAT` and press `[Enter ↵]`.
11. The file puts you in ReGIS mode and draws the box.

This is a simple series of ReGIS commands that can just as easily be entered directly at the ReGIS command line. But a more complicated ReGIS program could be written and debugged using this same method. Because the ReGIS command line does not allow you to edit your entries, changing a text file is easier than re-entering your commands on the ReGIS command line.

Mouse Support

The VT340 terminal supports locator devices with 2 to 4 buttons; a PC mouse can have 2 to 3 buttons. In order to provide full functionality, Reflection permits keystrokes to be mapped as mouse buttons (and vice versa).

The default mapping for these VT functions in Reflection 4 is shown in the keystroke definitions below; the default keystroke is on the left and the function name is on the right.

	<code>button-1-down</code>	=	<code>vt-button-1-down</code>
	<code>button-1-up</code>	=	<code>vt-button-1-up</code>
	<code>button-2-down</code>	=	<code>vt-button-2-down</code>
	<code>button-2-up</code>	=	<code>vt-button-2-up</code>
	<code>button-3-down</code>	=	<code>vt-button-3-down</code>
<code>alt</code>	<code>button-1-down</code>	=	<code>vt-button-3-down</code>
	<code>button-3-up</code>	=	<code>vt-button-3-up</code>
<code>alt</code>	<code>button-1-up</code>	=	<code>vt-button-3-up</code>
<code>alt</code>	<code>button-2-down</code>	=	<code>vt-button-4-down</code>
<code>alt</code>	<code>button-2-up</code>	=	<code>vt-button-4-up</code>

The functions `vt-button-3-down` and `vt-button-3-up` are mapped to `alt button-1-down` and `alt button-1-up` (for users with a 2-button mouse), and to `button-3-down` and `button-3-up` (to accommodate users with a 3-button mouse).

For more information:

- ▲ See "Keyboard Mapping," which begins on page 5, for instructions on how to remap these functions to different keystrokes.
- ▲ See page 140 for instructions on how to use a mouse in ReGIS graphics.

Introduction to ReGIS Commands

This chapter covers the conventions, syntax, and commands of the 10 basic ReGIS commands. A brief explanation, syntax, and example are provided for each one.

Conventions

The following conventions apply to ReGIS commands as they are presented in this manual:

- ▲ When you enter commands, you can use upper or lowercase letters. Uppercase letters are used here for consistency.
- ▲ Angle brackets (< >) enclose a description of the type of information required. The brackets are *not* part of the syntax.
- ▲ The expression [X, Y] indicates that you can select an absolute or relative position on the screen. X (horizontal) and Y (vertical) are variables for a coordinate position; the brackets *are* part of the syntax.
- ▲ Spaces are included between some parts of ReGIS commands for readability only; they are never required.
- ▲ When values are given for character display and unit cell sizes, the values assume a full-size, unscaled ReGIS display similar to the VT340 terminal. Reflection automatically scales characters to fit your actual display size.

Command Syntax

ReGIS command syntax is made up of relatively few elements. A complex image can be created by entering a straightforward, but rather long, series of commands. If you are creating a complex graphics image locally—that is, it is not being sent to you from the host—the most efficient way to do so is to create a text file with the ReGIS commands in it. This file can then be run at the Reflection command line with the command `TYPE <filespec>`. See the sample session on page 70.

Command Key Letter

Each ReGIS command is identified by a single letter, its *command key letter*. The *vector* command, for example, is written simply as V. Selected options and arguments follow the command.

For instance, the following sequence draws a line from the current position to a given pair of coordinates; the vector command is followed by its arguments:

```
V[300,400]
```

Until a new command key letter is given, all arguments apply to the current command. In the following example, the graphics cursor is positioned at [200,300], a vector is drawn from there to [400,100], and another vector is drawn from [400,100] to [200,50]. The last pair of coordinates uses the vector command, since it was the last command specified:

```
P[200,300] V[400,100] [200,50]
```

ReGIS Command Summary

The following is a brief summary of the command key letters. Each command is discussed in more detail beginning on page 79.

Command Key Letter	Command Name	Description
P	Position	Positions the graphics cursor without doing any drawing.
V	Vector	Draws straight lines.
C	Curve	Draws circles, arcs, and curves.
W	Write	Provides control over such elements as pen color and fill patterns.
F	Polygon Fill	Fills in a closed figure.
T	Text	Controls character display.
S	Screen	Controls elements such as background intensity and erasing.

Command Key Letter	Command Name	Description
L	Load	Defines and loads alternate character sets displayed with the text command.
R	Report	Provides information about the current status of ReGIS operations and is used to enter graphics input mode.
@	Macrograph	Stores and recalls ReGIS command strings.

Parentheses

Parentheses enclose options and suboptions of ReGIS commands. The following shows the screen command with the erase option:

S (E)

In the next example the cursor is positioned, and a circle with a radius of 150 pixels is drawn:

P[200,200] C(W(P2)) [+150]

The option in this case is the write command, and its suboption is the selection of a pattern for the line—P2—that consists of dashes. The pattern applies only to circles drawn subsequently by the curve command. See page 104 for available patterns.

Brackets

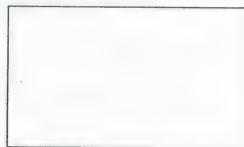
Brackets are used to enclose the following types of numeric values:

- ▲ Coordinate positions, both absolute and relative
- ▲ Height and width values (for text commands only)

By default, the ReGIS screen has the following coordinate system:

[0,0]

[799,0]



[0,479]

[799,479]

Coordinate System for ReGIS Screen

When specifying a coordinate position, follow these rules:

- ▲ Specify the X (horizontal) coordinate first, then the Y (vertical) coordinate. For example, [100,200] indicates an X coordinate of 100, and a Y coordinate of 200.
- ▲ If the Y coordinate for a new position is unchanged from its current value, you can omit the Y coordinate and just specify the X coordinate. For example, [200] indicates a new X coordinate, and an unchanged Y coordinate.
- ▲ If the X coordinate for a new position is unchanged from its current value, you can just use a comma and then specify the new Y coordinate. For example, [,50] indicates an unchanged X coordinate, and a new Y coordinate.
- ▲ A coordinate value beginning with a + or - sign indicates a value relative to the previous one. For example, [+100] indicates a new X coordinate 100 units to the right of the previous X coordinate. The Y coordinate is unchanged.

The following example indicates a position that is at the intersection of 200 on the X axis and 400 on the Y axis:

[200,400]

The next position is relative to the current one:

[+50,-25]

In the following example, X is relative and Y is absolute:

[+75,200]

Quotes

Quotes enclose the following ReGIS arguments:

- ▲ Text to display with text commands
- ▲ A character to use for polygon filling
- ▲ A character set name to select with the load command
- ▲ A character to use as an identifier for an argument to a load command

Either single quotes (') or double quotes (") can be used to define the beginning and end of the quoted argument; parentheses are not allowed.

Control Characters

ReGIS recognizes four control characters within a quoted string:

Character	Description	Keystroke
C _R	Carriage return	Ctrl - M
L _F	Linefeed	Ctrl - J
B _S	Backspace	Ctrl - H
H _T	Horizontal tab	Ctrl - I

If ReGIS encounters any control characters outside a quoted string, they are shown on the command line and ignored, with the exception of linefeed. When the linefeed control character is entered (the keystroke Ctrl-J), the command line is erased and you are positioned at the beginning of the line.

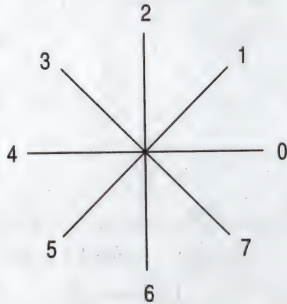
Resynchronization

A semicolon (;) in a command string instructs ReGIS to abort the current command and begin a new one. There are three instances in which the semicolon is not recognized as a resynchronization command:

- ▲ Within a quoted text string
- ▲ When it is part of a macrograph definition
- ▲ When it terminates a load command

Pixel Vector Directions

Many ReGIS positioning and drawing commands can use the pixel vector (PV) system, for moving and drawing one pixel at a time. This system uses eight directions at 45-degree intervals. Each direction is assigned a number:



Pixel Vector Directions

One PV number moves one pixel at a time in the direction specified. For example, the following command positions the cursor, then draws a line diagonally down to the right, then diagonally to the left. Each of the two lines is 10 pixels long:

```
P[100,100] V77777777775555555555
```

Moving and drawing one pixel at a time can be tedious. An option of the write command lets you give pixel vector points a multiplication factor, which specifies the number of times to repeat each point.

The following example sets a PV multiplier of 100 (the option `M100` to the write command) and then draws a box by listing the four numbers that represent the PV directions for down, left, up and right:

```
W(M100) V6420
```

Each side of the box is 100 pixels long.

Position Command

The position command moves the graphics cursor to an absolute or relative location on the screen. This chapter explains its options and arguments.

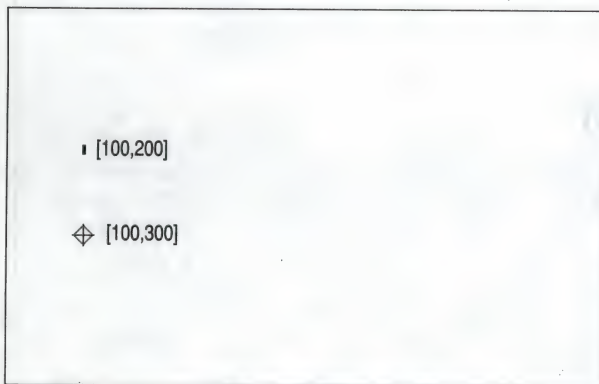
Cursor Positioning with [X,Y]

The cursor is positioned at the given X and Y coordinates. If the first value is omitted, its current value is assumed.

Format: P[X,Y]

In the following example, the second set of coordinates is assumed to be [100,300], even though the value 100 is not entered:

```
P[100,200] [,300]
```



Position Command—Cursor Positioning

In the next example, the second pair of coordinates is assumed to be [400,300] even though only the X coordinate is given. The following two expressions are equivalent:

```
P[100,300] [400,]
```

```
P[100,300] [400]
```


Relative values are entered using a plus or minus sign. The following command positions the cursor at [100,100] and then draws a vector 200 pixels down:

```
P[100,100] V[,+200]
```

When specifying X and Y coordinates, the X coordinate must always be first, and the Y coordinate must always be preceded by a comma. The default position is [0,0].

Cursor Positioning with Pixel Vectors

The cursor can be positioned with pixel vector values. The pixel vector directions are shown on page 78.

Format: P<pv>

In the following example, the cursor is moved one pixel at a time, 10 pixels to the right and 10 down:

```
P000000000006666666666
```

PV Multiplication

Using a temporary write control option, you can specify a multiplier for moving the cursor using the pixel vector system. The default is a multiplication factor of 1.

Format: P(W(M<n>)) <pv>

The value of <n> is the PV multiplier. For example, the next command uses a multiplier of 50 (M50) to move the cursor 50 coordinates for each PV value.

```
P(W(M50))6420
```

Begin Bounded Position Stack

With the *B* option—begin bounded position stack—the current position of the graphics cursor is saved. When the *E* option—end of position stack—is encountered, the cursor is returned to the saved position. Position, vector, curve, and other commands can be specified between the options that begin and end the bounded position stack. For each position stack begun with the *B* option, there must be a corresponding *E* option to end that stack.

Format: *P*(*B*) <options> (*E*)

In the following example, the cursor begins at [300,100]. This becomes the bounded position when the *B* option is specified. The cursor is then moved to two other positions on the screen. When the *E* is encountered, the cursor returns to [300,100]:

```
P[300,100] (B) [400,200] [300,150] (E)
```

Bounded positions can be nested up to 16 levels. The following example shows the command nested one level:

```
P[300,100] V(B) [+100] (B) [+100,+50]  
[-50,+100] (E) [+100,-100] [-50,+50] (E)
```

Start Unbounded Position Stack

The *S* option puts a dummy position on the stack. When the *E* option is encountered, the dummy stack position is ended, but the current cursor position remains the active one. The purpose for this option becomes clear in the discussion of the curve command on page 87.

Format: *P*(*S*) <options> (*E*)

Null Position Argument

This argument is equivalent to moving the cursor to a relative position of $[+0,+0]$. Its use is explained more fully in the discussion of the curve command's open curve option on page 91.

Format: `[]`

Vector Command

The vector command draws a line between the cursor position and the absolute or relative position that you specify. This chapter explains the available options and arguments for this command.

Draw Dot

By specifying the vector command with a null position argument, you can draw a single pixel at the current cursor position.

Format: `V[]`

The following commands position the cursor at [300,200] and draw a single pixel; the cursor does not move.

```
P[300,200] V[]
```

Draw Line

Specifying X and Y coordinates with the vector command, you can draw a straight line from the current position to an absolute or relative position.

Format: `V[X,Y]`

The following commands use absolute coordinates to draw a line from [200,100] to [250,300] and another one from [250,300] to [100,350]:

```
P[200,100] V[250,300] [100,350]
```

In the second pair of coordinates after the vector command, the V command is assumed and does not need to be entered. In the following example, a series of relative vector commands draws a pentagon after positioning the cursor at the absolute coordinates [200,100]:

```
P[200,100] V[+50,+50] [, +50] [-100] [, -50] [+50, -50]
```


Draw Lines Using Pixel Vectors

Using pixel vector numbers, you can draw a vector one pixel at a time from the current cursor position in any of the eight positions of the pixel vector system.

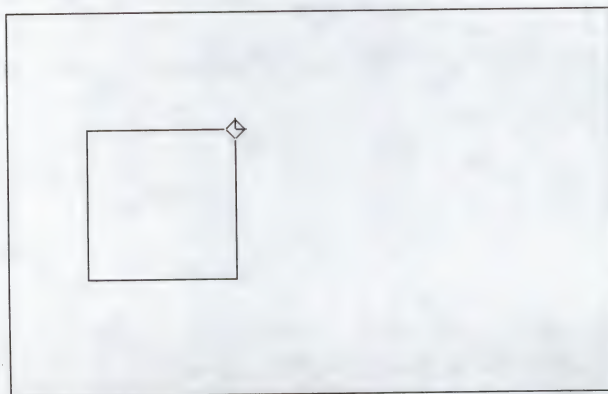
Format: `V<pv>`

The following example draws a line to the right that is ten pixels long:

```
V0000000000
```

With the write option given as an option to the vector command, a multiplication factor for pixel vectors can be designated, just as in the position command. The following example draws a square, each side of which is 200 pixels long:

```
P[300,150] V(W(M200))6420
```



Vector Command—Using Pixel Vectors

Begin Bounded Position Stack

When the `B` option (begin bounded position stack) is used, the current position of the graphics cursor is saved. When the `E` option (end stack) is entered, the cursor returns to the saved position. Other commands can be specified between the options that begin and end the bounded position stack.

Format: `V(B) <options> (E)`

The following draws a triangle:

```
P[100,100] V(B) [200,200] [50,250] (E)
```

The cursor begins at [100,100], and this position is made the start of a bounded position stack with the *B* option. A vector is then drawn to [200,200], then to [50,250], and the third side is drawn when *E* is entered.

Bounded position stacks can be nested up to 16 levels.

Start Unbounded Position Stack

The *S* option puts a dummy position on the stack. When the *E* option is encountered, the position stack is ended, no line is drawn, and the active position of the cursor does not change.

Format: `V(S) <options> (E)`

The unbounded stack option serves little use here; it is primarily for consistency with the stack options of the curve command discussed on page 87.

Temporary Write Control

This option gives you temporary control over the write command values used by the vector command; it ends when a new command is given. See page 103 for details about the write command.

Format: `V(W(<suboptions>))`

The following example temporarily defines *P4* as the line pattern and then draws a line using that pattern to a point 100 pixels down. When the second *V* command is given, the temporary write control is canceled, and the new line is drawn with the default pattern:

```
V(W(P4)) [, +100] V[+50, +100]
```


Curve Command

Curve commands can draw circles, arcs, and other types of curves. Temporary write controls are also available, so that curves can be drawn in a given pattern. This chapter explains the available options and arguments for this command.

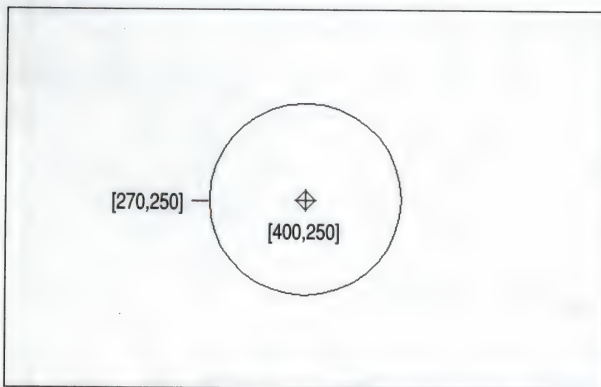
Center of Circle at Current Position

The coordinates $[X,Y]$ define a point on the circumference of the circle. The current cursor position is the circle's center.

Format: $C[X,Y]$

The following example positions the cursor at $[400,250]$ and then draws a circle whose circumference passes through $[270,250]$. The cursor position is $[400,250]$, the center of the circle:

$P[400,250] \ C[270]$



Curve Command—Center at Current Position

You can also use relative coordinates to specify the point to pass through. In effect, this specifies a radius for the circle. The following commands position the cursor at [200,300] and draw a circle that passes through a point 50 pixels to the right of the the cursor position; that is, with a radius of 50 pixels:

```
P[200,300] C[+50]
```

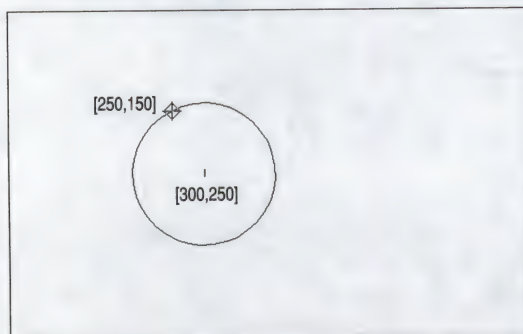
Center of Circle at Specified Position

The coordinates [X,Y] define the center of the circle while the current position defines a point on the circumference. This option remains in effect until a new command is given.

Format: C(C) [X,Y]

The following commands position the cursor at [250,150] and draw a circle whose center is at [300,250] and whose circumference passes through [250,150]:

```
P[250,150] C(C) [300,250]
```



Curve Command—Center at Specified Position

Arc with Center at Current Position

Using a command similar to the circle drawing command, you can draw an arc—a section of circle—with its center at the current cursor position and with a point on the circumference specified.

Format: C(A<degrees>) [X,Y]

The [X,Y] defines the starting point for the arc, in either absolute or relative coordinates. A is the arc option, and <degrees> determines the size and direction of the arc. If <degrees> is preceded by a minus sign, the direction of the arc is clockwise. If preceded by either a plus sign or no sign, the direction is counterclockwise. ReGIS draws arcs in 1 degree intervals, rounded to the closest integer degree.

The next example positions the cursor at [300,200] and draws a half circle counterclockwise starting 150 pixels to the right of the cursor position:

```
P[300,200] C(A180) [+150]
```

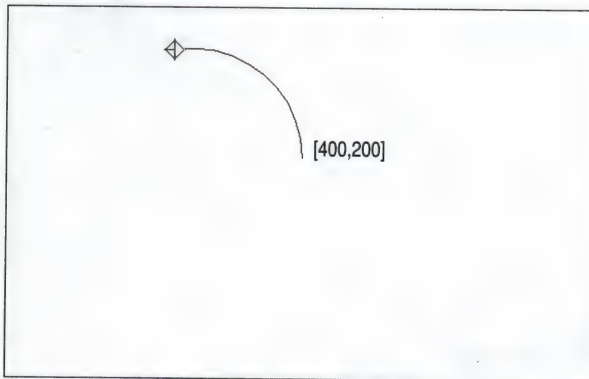
Arc with Center at Specified Position

This option lets you draw an arc that begins at the current position, and whose center is specified by [X,Y]. Using this option, you can link one arc to the next.

Format: C(A<degrees>C) [X,Y]

The following commands position the cursor at [400,200] and draw an arc counterclockwise 100 degrees. The center of the arc is 150 pixels to the left of the current cursor position:

```
P[400,200] C(A100C) [-150]
```



Curve Command—Arc Center Specified

Closed Curve Sequence

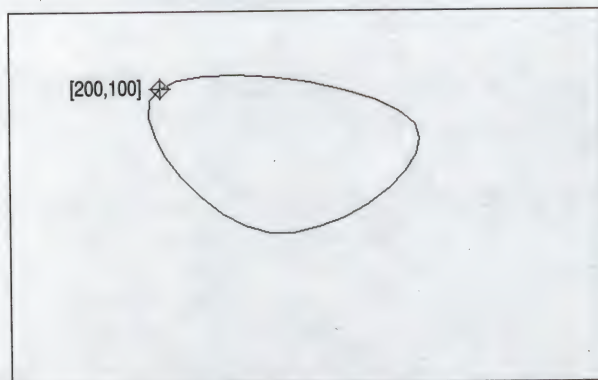
This command defines a closed curve based on a starting point, a minimum of two coordinates, and an endpoint; the curve is interpolated from the given coordinates. The *B* option indicates the start of the curve (similar to the option that starts a bounded position stack). When the sequence is ended with *E*, the cursor moves to the beginning position, drawing a line. Unlike the bounded position stack *B* and *E* options, which allow nesting, closed curve sequences can have only one *B* and one *E*.

Format: *C*(*B*) <*X,Y* coordinates> (*E*)

The *X* and *Y* coordinates can be absolute, relative, or a combination of the two. It may also include pixel vector values. Relative values are based on the last specified cursor position.

The following example draws an irregular, closed curve. Only two coordinates beyond the starting position are specified, and no lines are actually drawn until the *E* option ends the curve. This is because the curve drawing algorithm needs at least four positions to draw its curve. (The fourth point, or end point, is the same as the starting point.)

`P[200,100] C(B) [350,+200] [+200;-150] (E)`



Curve Command—Closed Curve Sequence

Open Curve Sequence

This option lets you define an open curve based on a starting point, a minimum of three coordinates, and an endpoint. The S option indicates the start of the open curve (similar to the unbounded position stack option). When the sequence is ended with E, the cursor remains in the last position that you specified. Unlike the unbounded position stack option, which allows nesting, an open curve sequence can have only one S and one E.

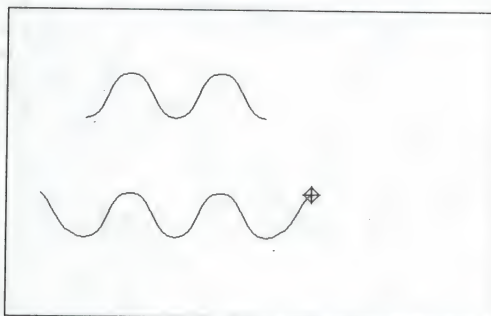
Format: C(S) <X,Y coordinates> (E)

To draw the curve completely, you need to include a null position argument, [], at the beginning and end; otherwise, ReGIS only draws from the first position to the next-to-last one.

The following two examples illustrate this point: the first is drawn without null position arguments and the second is the same curve, further down on the screen, with null positions.

```
P[50,100]
C(S)
[125,175] [200,100] [275,175] [350,100]
[425,175] [500,100]
(E)
```

```
P[50,300]
C(S) []
[125,375] [200,300] [275,375] [350,300]
[425,375] [500,300]
[] (E)
```



Curve Command—Open Curve Sequences

Temporary Write Control

When you draw curves, any write control options currently active remain active. With temporary write control options, you can temporarily change the write controls for the curve you are drawing. The temporary controls remain in effect until you specify another temporary write control, or until a new command key letter is given. Any of the options discussed with the write command on page 103 can be used.

Format: C(W(<suboptions>))

The following example draws an open curve similar to the one in the preceding figure, only now the P2 line pattern is used. (For a list of patterns, see page 104.)

```
P[50,300]
C(W(P2)) (S) []
[125,375] [200,300] [275,375]
[350,300] [425,375] [500,300] []
(E)
```

Screen Command

Using the screen command `S`, you can control attributes of the entire screen, including the background intensity, screen erasing, and the shape of the graphics cursor. This chapter explains the available options and arguments for this command.

Display Addressing

Normally, the screen has default coordinates of `[0,0]` for the upper-left corner and `[799,479]` for the lower-right corner. With the display address option, you can define the coordinates within which ReGIS images appear. If you create applications that are run on terminals other than the VT300 series, defining display coordinates can help make ReGIS images more portable. ReGIS automatically scales the images to fit correctly in the defined coordinates, so that squares appear square and angles appear at the proper angle.

Format: `S(A[X1,Y1] [X2,Y2])`

The first set of coordinates is for the upper left corner of the screen, and the second set is for the lower right corner.

Scrolling

With a scroll argument to the screen command, you can offset data on the display, but leave the coordinate system unchanged; the image moves relative to the screen origin `[0,0]`. The scroll argument can use absolute or relative coordinate values, a combination of the two, or pixel vector (PV) values.

Format: `S[X,Y]` (using coordinates)
`S<PV value>` (using pixel vectors)

If you use PV values, the current PV multiplier is used. You can also specify a temporary PV multiplier, using a temporary write control as an option to the screen command.

Format: `S(W(M<multiplier>))<PV value>`

The temporary write control stays in effect until a new command key letter is used, or another temporary write control is given.

Any data that is scrolled outside of the display boundaries is lost; the data cannot be recovered by reversing the scrolling.

Print Screen

This command option lets you print a hard copy of your screen. To configure the printer, see "Graphics Printer Setup" in the *Technical Reference*.

An image sent to the host is sent as a series of sixels, which can be collected in a host file or sent to another graphics terminal. There are three formats for the print command:

- ▲ Print the entire screen.

Format: S(H)

- ▲ Print a rectangular portion of the screen, from the cursor position to a given set of coordinates.

Format: S(H[X,Y])

- ▲ Print a rectangular portion of the screen, from one set of coordinates to another.

Format: S(H[X₁,Y₁] [X₂,Y₂])

You can add a print offset to this command so the image is printed X number of pixels from the left printing margin and Y number of pixels from the top printing margin. The default offset is [50,0]. If you define a new offset, the new value remains in effect until redefined. Define an offset as follows:

Format: S(H(P[X,Y]))

You can also combine the print offset with the print command:

Format: S(H(P[X,Y]) [X₁,Y₁] [X₂,Y₂])

Output Mapping Values

The VT240, VT241, and VT330 use a 2-bit code to represent each screen pixel, providing up to 4 different shades of gray (or colors in the case of the VT241). The VT340 uses a 4-bit code to represent each pixel, providing up to 16 different colors.

Each color is maintained in an *output map location*, a location in memory that stores a set of values that define the color. The VT240, VT241, and VT330, each with a 2-plane bitmap, have 4 output map locations, while the VT340, with a 4-plane bitmap, has 16 output map locations.

Using output mapping controls, you can change the colors stored in specific output map locations, letting you change the color of a ReGIS image without redrawing it.

The output map locations for ReGIS graphics are linked to the Graphics Color Palette Setup dialog box. You can use this dialog box to manually change the colors in the output map.

When an application changes ReGIS colors (using the mapping control options explained here), the new colors are shown in the Color Map SetUp. If the application does not restore the original colors after changing them, the color of text attributes that use those map locations will be changed as well. See the *Technical Reference* for more information about the Graphics Color Palette Setup dialog box.

Specifying Colors

Each color in the output map can be specified in one of two ways:

- ▲ Hue, lightness, and saturation (HLS) values: The hue can range from 0 to 360 degrees, lightness from 0 to 100 percent, and saturation from 0 to 100 percent.
- ▲ RGB color abbreviations: Eight single-letter abbreviations represent eight different colors. Some colors are secondary colors made by mixing red, green, and blue.

Letter	Color	Letter	Color
D	Black (dark)	C	Cyan
R	Red	Y	Yellow
G	Green	M	Magenta
B	Blue	W	White

Monochrome Graphics

The *L*, or *lightness*, parameter of an HLS value defines a color's monochrome shade. For the VT240 and VT330 monochrome terminals, only the lightness parameter has meaning; saturation is always 0 for gray shades, and hue has no effect. If a full HLS specification is given (as explained in the next section), the hue and saturation are ignored. If RGB abbreviations are used with a monochrome terminal type, the lightness is set to 50 for all color abbreviations except D (which has a lightness of 0) and W (which has a lightness of 99).

When the terminal type is VT240 or VT330, the following command selects a monochrome shade for an output map location. The appropriate color on the Graphics Color Palette Setup dialog box is updated to show the new monochrome shade.

Format: S(M<0 to 3>(L<value>))

This command can also be used when the terminal type is VT241 or VT340; however, the color stored at the selected output map location is replaced by the specified monochrome shade (see page 97 for more information).

The following table lists the monochrome output map locations for the VT240 and VT330 terminals and their default HLS and equivalent RGB values.

VT240 and VT330 Default Monochrome Mappings

Map Location	Shade	HLS Values			Equivalent RGB Values		
		H	L	S	R	G	B
0	Black	0	0	0	0	0	0
1	Dark gray	0	33	0	33	33	33
2	Light gray	0	66	0	66	66	66
3	White	0	100	0	100	100	100

Color Graphics

When Reflection is configured for VT241 graphics emulation, there are 4 output map locations. For VT340 graphics, there are 16 output map locations. Each location stores a monochrome shade and a color, and the two can be changed together or separately.

Changing Monochrome Shades

As mentioned in “Monochrome Graphics” on page 96, specifying a lightness value alone changes the monochrome shade of an output map location.

Format: S(M<0 to 3>(L<value>)) (VT241)

S(M<0 to 15>(L<value>)) (VT340)

To change the lightness of multiple output map locations at once, you can specify multiple output maps and lightness values in the same command. For example, the following changes the monochrome shades stored in output map locations 1, 2, and 4:

S(M1(L35)2(L45)4(L79))

Changing Color Values

You can change the color in an output map location by specifying a new color using either an HLS or RGB value. You can use either of the following two commands to select a new color value for an output map location. HLS values and RGB values should not be combined in one command. When you change colors, the appropriate color on the Graphics Color Palette Setup dialog box is updated to show the change.

Format: S(M<0 to 3>(H<hue>L<lightness>S<saturation>)) (VT241)

S(M<0 to 15>(H<hue>L<lightness>S<saturation>)) (VT340)

Format: S(M<0 to 3>(<RGB letter>)) (VT241)

S(M<0 to 15>(<RGB letter>)) (VT340)

In the following example, the color in output map location 0 is changed to blue, using an RGB value:

S(M0(B))

To change the colors of multiple output map locations at once, you can specify multiple output map locations and color values in the same command. For example, the following command changes the colors in output maps 2 and 7, using HLS values:

S(M2(H300 L75 S60) 7(H180 L50 S100))

Remember that the VT241 and VT340 store both a monochrome and a color value for each output map location. This means that when you change a color using one of the commands above, the monochrome shade stored at that map location is changed to a shade of gray corresponding to the lightness of the color; if a monochrome value was already defined for that map location, it is replaced by the new monochrome shade.

For example, the following commands define a monochrome value for output map location 12, then define a color for the same location using HLS values:

```
S(M12(L33))
S(M12(H120 L46 S71))
```

The first command defines a shade of gray with a lightness value of 33. The second command, which defines red on the VT340, would specify a shade of gray with a lightness of 46 on a monochrome terminal; the value of 33 is overwritten.

You can change a color in the output map while retaining the monochrome value stored in the same location by using the following commands:

```
Format: S(M<0 to 3>(A <HLS or RGB values>)) (VT241)
        S(M<0 to 15>(A <HLS or RGB values>)) (VT340)
```

The A suboption specifies that only the *color* at the specified map location should be changed; the monochrome value should remain as it is.

Using the previous example, you could keep the monochrome shade at a lightness value of 33, and change only the color value to red with the A suboption. The two commands would then look like this:

```
S(M12(L33))
S(M12(AH120 L46 S71))
```

The A suboption can be useful when programming ReGIS graphics for different types of terminals, such as the VT330 and VT340. On the VT330, you would want the exact lightness value you specify, rather than a lightness value corresponding to the lightness of a color. On the VT340, however, you would want to show the actual color. Using the A suboption solves this problem.

The default color maps for the VT241 and VT340 terminals are shown in the tables below.

VT241 Default Color Map

Map Location	Shade	HLS Values			RGB Values		
		H	L	S	R	G	B
0	Black	0	0	0	0	0	0
1	Blue	0	50	60	20	20	80
2	Red	120	46	71	80	13	13
3	Green	240	50	60	20	80	20

VT340 Default Color Map

Map Location	Shade	HLS Values			RGB Values		
		H	L	S	R	G	B
0	Black	0	0	0	0	0	0
1	Blue	0	50	60	20	20	80
2	Red	120	46	71	80	13	13
3	Green	240	50	60	20	80	20
4	Magenta	60	50	60	80	20	80
5	Cyan	300	50	60	20	80	80
6	Yellow	180	50	60	80	80	20
7	Gray 50%	0	53	0	53	53	53
8	Gray 25%	0	26	0	26	26	26
9	Blue	0	46	28	33	33	60
10	Red	120	43	38	60	26	26
11	Green	240	46	28	33	60	33
12	Magenta	60	46	38	60	33	60
13	Cyan	300	46	28	33	60	60
14	Yellow	180	46	28	60	60	33
15	Gray 75%	0	80	0	80	80	80

Background Intensity

To select a shade or color for the screen background, you can either give an output map location or provide an RGB or HLS value. The three formats follow:

Format: `S(I<n>)`
`S(I(<RGB>))`
`S(I(H<n>L<n>S<n>))`

With the second and third methods, the output map location that contains the color closest to the one you specify is selected. For example, if Reflection is configured as a VT241, which has four colors, entering `S(I(M))` gives you the color closest to magenta; on the VT241, this gives you red (unless the output map was changed from its defaults).

Selecting a background intensity does not change the color stored at the output map location.

Time Delay

You can delay the execution of a ReGIS command by specifying a time delay. The number you give is measured in ticks: 60 ticks equal 1 second. The maximum time you can specify as a delay is 32,767 ticks, which is equal to about 9.1 minutes.

Format: `S(T<0 to 32767>)`

Screen Erase

You can erase the entire screen to the background screen color.

Format: `S(E)`

This command does not change the color stored in any output map locations, and it does not move the cursor. You can combine the screen erase option with a command to change the background intensity. The comma in the command is required.

Format: `S(I<value>,E)`

Graphics Cursor

ReGIS uses two types of graphics cursors: a graphics *output* cursor and a graphics *input* cursor.

- ▲ The output cursor appears when ReGIS is waiting for commands from the host (or from the ReGIS command line). You control whether the output cursor is displayed and what shape it should have with the ReGIS commands described below.
- ▲ The input cursor appears when ReGIS is waiting for graphics input; for example, when the host requests a cursor position report (see the control function section of the *Technical Reference*). You can move the input cursor with the mouse, and change its shape with a screen command option.

Output Cursor Control

Use a screen command option to control whether or not the graphics output cursor is displayed.

Format: S(C<0 or 1>)

A 0 turns the output cursor off; a 1 turns it on.

The style of the output cursor is determined by a suboption.

Format: S(C(H<0 to 2>))

If the number is omitted, or is 0 or 1, the cursor style is a diamond (the default).

If the number is 2, the cursor appears as a crosshair.

Input Cursor Control

The graphics input cursor is typically used to determine the current cursor position with the report command (see page 137 for details about the report command).

The style of the input cursor can be selected from five predefined styles, or defined by the user.

Format: S(C(I<0 to 4>)) (predefined styles)
S(C(I[X,Y]"<characters>")) (user-defined input cursor)

The table below lists the predefined input cursor styles:

Input Cursor Styles

Number	Input Cursor Style
0	Crosshair (default)
1	Diamond
2	Crosshair
3	Rubberband line
4	Rubberband rectangle

A user-defined cursor is composed of two characters: one is displayed in the foreground shade or color, and the other is displayed in the background shade or color. Monochrome graphics emulation uses output map location 2 for the foreground and 1 for the background. The color graphics emulation uses location 14 for the foreground and 1 for the background.

The size of the input cursor is limited to 16×24 pixels. The following example defines an input cursor that consists of a vertical bar inside of an uppercase O: ⓪

```
S(C(I[+4,+10]"OI"))
```

The parameter [+4,+10] selects the coordinate for the origin of the cursor.

The two characters you use to compose the cursor can come from either the built-in character sets or from characters you design and load. See the description of the load command on page 133 for more information. You can also use the text command (described on page 121) to change the size of the character.

Write Command

The write command gives you control over patterns, the foreground color, four types of writing, plane selection, and shading. When used alone, a write command affects all commands that follow it, until another write command is encountered. If used as a temporary write control, the *W* is enclosed in parentheses and it becomes a temporary option of the command key letter. For example, in the following sequence, the write command becomes an option of the vector command and has temporary effect:

```
V[+100] (W(E)) [+100]
```

The control remains in effect until another command is issued.

When you use the write command to control how vectors, curves, and shaded areas appear, you typically set three parameters first:

1. Select a pattern using the *P* option.

The pattern can include a multiplication factor, specified with the *M* suboption.

2. Select a foreground color using the *I* option.
3. Select a write mode: overlay, replace, complement, or erase.

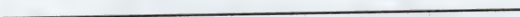
When an image is drawn, these and other parameters combine to control its appearance.

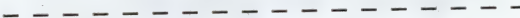
Select Standard Pattern

The *P* option selects one of ten standard patterns for drawing; the patterns are shown in the figure below. The default pattern is *P1*.

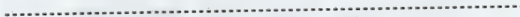
Format: *W*(*P*<0 to 9>)

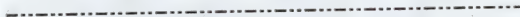
W(*P*0)

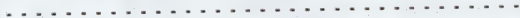
W(*P*1) 

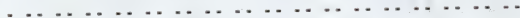
W(*P*2) 

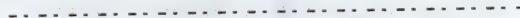
W(*P*3) 

W(*P*4) 

W(*P*5) 

W(*P*6) 

W(*P*7) 

W(*P*8) 

W(*P*9) 

Write Command—Standard Line Patterns

The following example selects line pattern 5, and draws a vector 100 pixels long:

```
W(P5)
V[+100]
```

Select Binary Pattern

This option lets you define your own 8-bit writing pattern.

Format: *W*(*P*<binary pattern>)

The binary pattern is made up of 2 to 8 bits that are either on or off. If you specify fewer than 8 bits, ReGIS repeats as much of the pattern as possible in what remains of the 8-bit segment. If more than 8 bits are specified, only the last 8 make up the defined pattern.

For example, both of the following two binary pattern selections produce the same dotted pattern (identical to the standard pattern 4):

```
W(P01)
W(P01010101)
```

Both standard and binary patterns can be multiplied, letting you create writing patterns longer than 8 bits. Using a multiplication factor as a suboption to the P option, you specify the number of times each bit in the 8-bit pattern should repeat. The default multiplication factor is 2.

Format: W(P<pattern>(M<1 to 16>))

For example, when the standard pattern 4 is multiplied by a factor of 4, the bit pattern changes from 10101010 to 11110000111100001111000011110000. The command that creates this pattern looks like this:

```
W(P4(M4))
```

In the following example, the binary pattern specified by W(P01) is multiplied by five, and a vector 100 pixels long is drawn:

```
W(P01(M5))
V[+100]
```

Pixel Vector Multiplication

The M option to the W command lets you assign a multiplication factor to the magnitude of the pixel vectors used for writing.

Format: W(M<multiplication factor>)

With the default multiplication factor of 1, the following command draws four lines that are each two pixels long. The numbers given are the pixel vector directions (see page 78).

```
V00664422
```

The next example includes a multiplication factor of 100; this time the four lines are each 200 pixels long.

```
W(M100) V00664422
```


Foreground Intensity

This option determines the shade or color of the foreground image. (A similar option of the screen command determines the background color; see page 95.) On the VT240, VT241, and VT330, up to 4 shades (or colors for the VT241) are available. On the VT340, up to 16 shades or colors are available. There are three ways to define the shade or color:

- ▲ An output map location can be given, meaning that whatever color is currently stored in that location is used. Output mapping is explained in the discussion of the screen command on page 95.

Format: W(I<0 to 3>) (VT240, VT241, or VT330)
 W(I<0 to 15>) (VT340)

- ▲ An RGB abbreviation can be used to select an output map location. The output map location containing the color closest to the one you specify is selected.

Format: W(I(<RGB>))

The following RGB colors are available:

Letter	Color	Letter	Color
D	Black (dark)	C	Cyan
R	Red	Y	Yellow
G	Green	M	Magenta
B	Blue	W	White

- ▲ A hue, lightness, and saturation (HLS) value for the color can be given. The HLS value selects the output map location containing the color closest to the one you specify.

Format: W(I(H<n>L<n>S<n>))

Hue values can range from 0 to 360; blue has a hue value of 0, red a value of 120, and green a value of 240. Lightness and saturation values can range from 0 to 100.

The selected foreground intensity is used only for the arguments that follow the option. This way, different parts of the image can have different colors.

Writing Styles

The write command has four writing styles: overlay (the default), replace, complement, and erase. Each one affects the way ReGIS draws images into graphics memory.

Writing Style	Command	Effect
Overlay	W(V)	Foreground where pattern is on; no change where pattern is off.
Replace	W(R)	Background where pattern is off; foreground where pattern is on.
Complement	W(C)	Where pattern is on, the complementary pixel values are used.
Erase	W(E)	If negative pattern control is off, writing is done in the background color, regardless of the pattern. If negative pattern control is on, writing is done in the foreground color.

Only the bitmap planes that are enabled, as defined by the plane select option described on page 115, are affected by each writing style. The following sections explain more about each style.

Overlay Writing

When overlay writing is selected, pixels are drawn with the foreground color where bits in the writing pattern are on (1). Bits in the pattern that are off (0) are ignored. Overlay writing is the default style; you do not need to specify an option to select it, unless you want to change from another style.

Format: W(V)

In the following example, the display is cleared with a screen command (screen controls are explained on page 93) and the cursor is positioned to [6,200]. A series of write commands select yellow for the foreground, overlay writing, and line pattern P2 (a series of dashes) with a multiplication factor of 1. The vector command is then used to draw a line 100 pixels long.

```
S(I0,E)
P[6,200]
W(I(Y)) (V) (P2(M1))
V[+100]
```



Replace Writing

When replace writing is selected, pixels are drawn with the foreground color where bits in the writing pattern are on (1). Pixels are drawn with the background color where bits in the pattern are off (0). (If the background color is not changed from its default using a screen command, the effect of replace writing appears the same as for overlay writing.)

Format: W(R)

In the following example, the display is first cleared and the cursor positioned as in the overlay writing example. Then, the background color is set to the color in output map location 3, the foreground color to output map location 6, and replace writing is selected. Line pattern 2 (dashes) is selected with a multiplication factor of 1, and finally a vector 100 pixels long is drawn.

```
S(I0,E)
P[6,200]
S(I3)
W(I6) (R)
W(P2(M1))
V[+100]
```



Complement Writing

When complement writing is selected, the binary value of the pixel's color (that is, the value of its output map location) is complemented where bits in the writing pattern are on (1). Bits in the pattern that are off (0) are ignored.

For example, on the VT240 terminal (with a 2-bit code for each pixel), a pixel with a color value of 1 (binary 01) has a complemented value of 2 (binary 10), and complement writing is done using the color in output map location 2.

Likewise, on the VT340 terminal (with a 4-bit code for each pixel), a pixel with a color value of 7 (binary 0111) has a complemented value of 8 (binary 1000), and complement writing is done using the color in output map location 8.

The actual color that appears on the screen is not necessarily the true complementary color of the original color. The setting of the foreground intensity (the (I) option) has no effect on complement writing.

Format: W(C) (specifies complement writing alone)
 W(F<plane code>,C) (specifies complement writing and a plane selection)

In the following example, the first two lines clear the display and position the cursor. Then, the screen color is changed to the color in output map location 2, replace writing is selected, using the color in output map location 5 and line pattern 2 with a multiplication factor of 1, and a vector 100 pixels long is drawn. Finally, the cursor is repositioned, complement writing is selected with line pattern 1 (solid), and another vector 100 pixels long is drawn. Complement writing causes the new line to be drawn using the colors in output map locations 13 and 10.

```
S(I0,E)
P[6,100]
S(I2)
W(R,I5,P2(M1))
V[+100]
P[6,100]
W(C,P1)
V[+100]
```



Erase Writing

With erase writing, all pixels are drawn using the background color, regardless of whether the writing pattern bits are on or off. If negative pattern control is turned on (as described below), erase writing draws using the foreground color.

Format: W(E)

In the following example, the screen is cleared and the cursor positioned. Then the background color is set to the color in output map location 6, erase writing is set, line pattern P2 is selected with a multiplication factor of 1, and a vector 100 pixels long is drawn.

```
S(I0,E)
P[6,200]
S(I6)
W(E)
W(P2(M1))
V[+100]
```

Negative Pattern Control

When negative pattern control is turned on with W(N1), the current writing pattern is reversed. By default, this option is off: W(N0).

Format: W(N<0 or 1>)

Shading

The shading option lets you shade the inside of an image in ReGIS as it is drawn. The current writing pattern determines the pattern for shaded areas, and the current writing style determines how 1 bits and 0 bits appear.

Shading occurs from the point being drawn by the vector or curve command to a *shading reference line*. The default shading reference line is a horizontal line defined by the Y coordinate of the current cursor position.

Format: W(S<0 or 1>)

The option (S1) turns shading on. By default, shading is off.

Selecting a Horizontal Reference Line

You can select a new horizontal reference line by specifying a Y coordinate as an argument to the shading option:

Format: `W(S[,Y])`

The following example draws a filled, bullet-shaped figure. The cursor is positioned, shading is turned on, a horizontal reference line 300 pixels down is defined, and a curve with a radius of 100 pixels is drawn at the top. After the commands are executed, the cursor remains at [125,125].

```
P[125,125]
W(S1[,+300])
C[+100]
```

Selecting a Vertical Reference Line

You can also select a vertical reference line.

Format: `W(S(X) [X])`

The next series of commands draws another filled, bullet-shaped figure, this time using a vertical reference line as the reference for shading.

```
P[600,375]
W(S1(X) [-340])
C[+50]
```

Selecting a Character for Shading

You can select a given character to be used for shading.

Format: `W(S'<character>')`

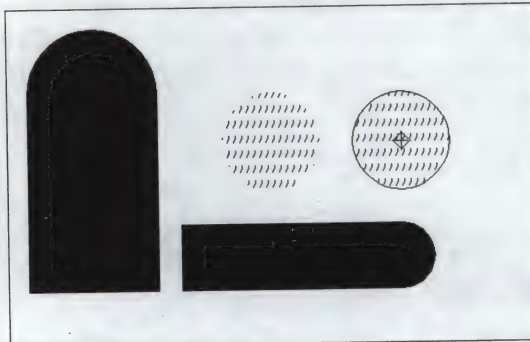
The following example positions the cursor, selects a forward slash (/) as the character to be used for shading, and draws a filled circle. Selecting a character for shading also turns shading on.

```
P[400,200]
W(S'/'')
C[+75]
```


To outline the shaded circle above, turn shading off and draw the circle again. You can do this by adding the following commands:

```
W(S0)
C(W(I2)) [+75]
```

The following figure shows all four of the shading command examples described.



Write Command—Shading Options

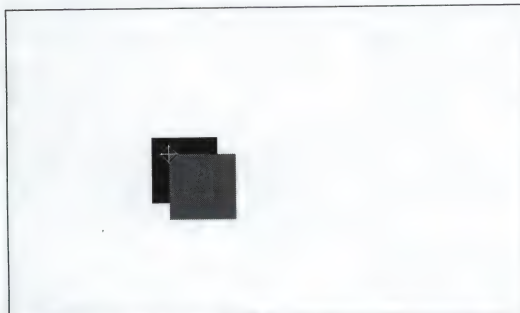
Examples of Writing Styles

The following are detailed examples of each style of writing. The polygon fill command *F* (explained on page 117) is used to draw each square.

```

S(I15,E)
W(I0)
P[225,200]
F(V[+100] [,+100] [-100] [-100])
W(I7,V)
P[250,225]
F(V[+100] [,+100] [-100] [-100])

```

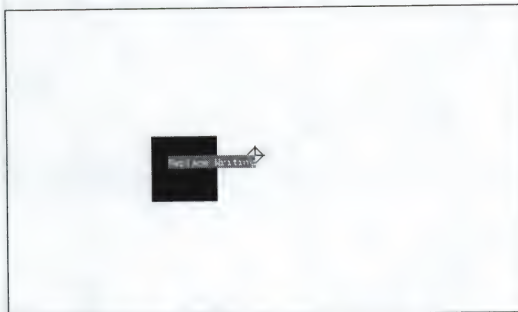


Write Command—Overlay Writing

```

S(I15,E)
W(I0)
P[225,200]
F(V[+100] [,+100] [-100] [-100])
S(I7)
W(I15,R)
P[250,230]
T 'Replace Writing'

```

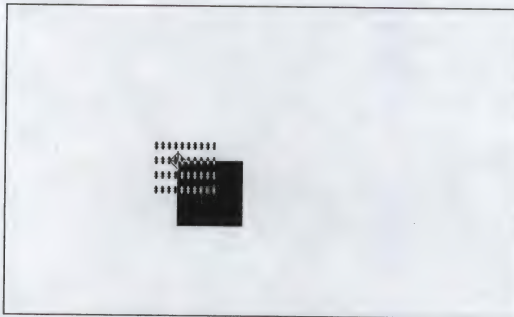


Write Command—Replace Writing


```

S(I15,E)
W(I0)
P[225,200]
W(S'#')
F(V[+100] [, +100] [-100] [, -100])
W(S1S0,I15)
P[268,240]
W(C)
F(V[+100] [, +100] [-100] [, -100])

```

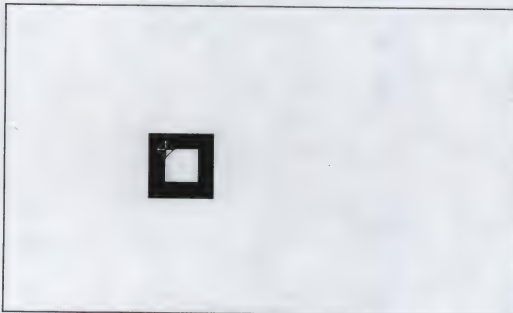


Write Command—Complement Writing

```

S(I15,E)
W(I0)
P[225,200]
F(V[+100] [, +100] [-100] [, -100])
W(E)
P[250,225]
F(V[+50] [, +50] [-50] [, -50])

```



Write Command—Erase Writing

Plane Selection

When Reflection's terminal ID is VT240, VT241, or VT330, ReGIS can write to 2 bitmap planes. When the terminal ID is VT340, 4 bitmap planes are available. For the VT240, VT241, and VT330, each pixel has a 2-bit code—one bit for each plane—giving up to four different simultaneous shades of gray (or colors in the case of the VT241). For the VT340, each pixel has a 4-bit code, allowing for 16 simultaneous colors.

With the plane selection option to the write command, you can specify which bitmap planes can be written to. This lets you disable or enable each plane individually. This can affect how complement writing appears, how images are overlaid with one another, and so on.

Format: W(F<0 to 3>) (VT240, VT241, or VT330)
W(F<0 to 15>) (VT340)

The code numbers that select the bitmap planes (0 to 3 or 0 to 15) correspond to the binary representations of the planes you can write to. A bit that is on (1) selects a plane for writing.

For example, with the VT330 terminal, the digit 2 in the command W(F2) corresponds to the binary code 10, allowing you to write to bitmap plane 1 (counting from 0, right to left). With the VT340 terminal, the digit 7 in the command W(F7) corresponds to the binary code 0111, letting you write to planes 0, 1, and 2; plane 3 is left disabled.

VT240, VT241, and VT330 Writing Planes

Command	Planes that can be written to
W(F0)	None
W(F1)	0
W(F2)	1
W(F3)	All

VT340 Writing Planes

Command	Planes that can be written to
W(F0)	None
W(F1)	0
W(F2)	1
W(F3)	0 and 1
W(F4)	2
W(F5)	0 and 2
W(F6)	1 and 2
W(F7)	0, 1, and 2
W(F8)	3
W(F9)	0 and 3
W(F10)	1 and 3
W(F11)	0, 1, and 3
W(F12)	2 and 3
W(F13)	0, 2, and 3
W(F14)	1, 2, and 3
W(F15)	All

After drawing with a restricted plane selection, you should restore writing to all planes with the command W(F15).

Polygon Fill Command

The polygon fill command *F* is used to draw and fill closed figures such as squares, triangles, and circles. There are four options to the polygon fill command, letting you position the cursor, draw vectors, draw curves, and specify temporary write controls. In general, the fill command accepts any ReGIS commands as options (enclosed in parentheses). The current foreground color specified by a screen or write command is used as the color for polygon filling.

Vector

All of the options and arguments for the vector command (page 83) can be used as suboptions to the fill command.

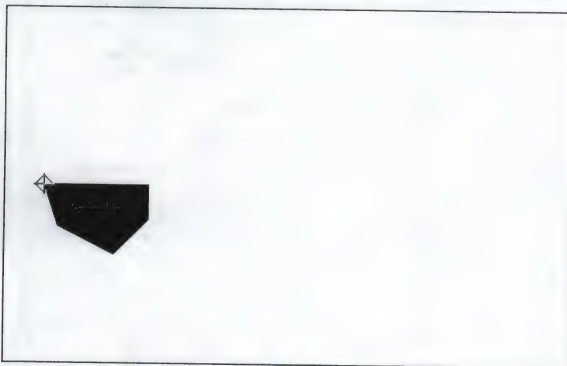
Format: *F*(*V*<coordinates>)

The following example creates an irregular, filled, 5-sided figure:

```
P[50,250]
```

```
F(V[200,250] [,+50] [-50,+50] [-80,-40] [-20,-60])
```

Because there is no way for ReGIS to know when the polygon is complete, it is not drawn and filled until the closing parenthesis is entered.



Polygon Fill Command—Vector Option

Curve

The options and arguments for the curve command (page 87) can all be used as suboptions to the fill command.

Format: F(C<coordinates>)

The following creates a filled ellipse. The B and E options are used to define the beginning and end of the curve.

```
P[150,200]
F(C(B) [+300] [, +200] [-300] (E))
```

To draw a filled circle with its center at [400,200] and a radius of 100 pixels, use the following commands:

```
P[400,200]
F(C[+100])
```

Position with Curve

This option is used to connect curves, arcs, and vectors.

Format: F(C(A+<degrees>) <position 1> P<position 2>)

The C identifies the curve option, and the A+<degrees> parameter indicates the arc suboption. Both are explained on page 88.

The next example produces a slanted, rectangular shape with rounded ends. An initial position is specified at [200,200]. Then the fill command is given, first defining a 90 degree counterclockwise arc that starts 50 pixels right and 50 pixels up from the current position. The current position is then moved down and right 100 pixels, and a second 90 degree arc is defined from a point 50 pixels left and 50 pixels down from the new position. When the closing parenthesis is encountered, the shape defined by the coordinates is filled.

```
P[200,200]
F(C(A+90) [+50,-50]
P[+100,+100]
C(A+90)
[-50,+50])
```




Polygon Fill Command—Position Option

Temporary Write Control

Temporary write control can be used either as an option of the fill command or as a suboption of the curve and vector options. As an option of the F command, the syntax is as follows:

Format: F(W(<suboptions>) <options>)

Any of the options for the write command can be used in the parameter W(<suboptions>). In the format above, the temporary write control applies only to the <options> of the polygon fill command.

The syntax for using the temporary write control as a suboption of the C or V options is as follows:

Format: F(C(W(<sub-suboptions>) <suboptions>) <options>)

In this format the W(<sub-suboptions>) parameter only has an effect on the curve command and its suboptions; it does not necessarily affect the fill command.

Text Command

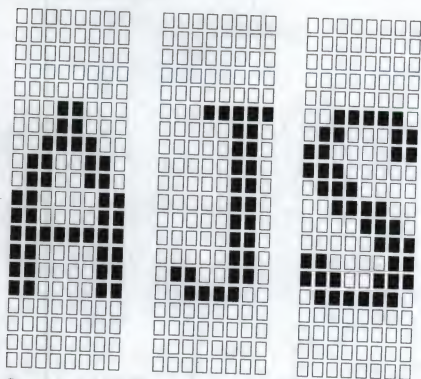
The text command *T* determines the size and orientation of graphics text characters. Options of the text command let you select a character set, the spacing between characters, the size and height of characters, and so on. Until you define new text characteristics, the ones you select remain in effect, unless they have been defined with temporary write controls.

In discussing characters cells and sizes, this manual uses values corresponding to a full-sized, unscaled ReGIS display of 800×480 pixels, similar to the VT340 terminal. Reflection automatically scales characters to fit the size of the display.

About Character Formats

When you use the text command, ReGIS selects each character from a stored character set and draws it to the screen, scaling and orienting the character according to the values defined. This lets you align subsequent characters using PV values and position commands.

Each character is drawn with the upper-left pixel of the character cell at the cursor position. The following figure shows a few sample characters from the ASCII character set.



Sample ASCII Characters: 8 x 20 Character Cell

When characters are drawn normally—without any tilt—each character appears below and to the right of the cursor. If a 180 degree tilt is applied, characters appear upside-down, with each one starting above and to the left of the cursor. The upper-left pixel of the character cell acts as a “pivot” around which the character cell is rotated.

Knowing information about the size and orientation of character cells can be useful when working with the size and tilt options of the text command, explained on pages 126 and 128.

About Text Formats

The following conventions are used to specify text strings in ReGIS:

- ▲ Single or double quotes can be used to mark the beginning and end of the string. The same type of marks must delimit the string.

For example, "this is text" and 'this is text' are equivalent.

- ▲ If one type of quote is used to delimit the string, the other type can be used within the string.

For example:

"it's a Weimaraner"	appears as	it's a Weimaraner
'"Wow," she exclaimed'	appears as	"Wow," she exclaimed

- ▲ Two quote marks in a row, with no space separating them, are used to include both types of quotes within a string.

For example:

"Quote mark: "" "	appears as	Quote mark: "
'it''s a girl'	appears as	it's a girl

- ▲ A comma placed between two strings that have the same type of quote marks combines the strings.

For example:

"Press ", "enter"	appears as	Press enter
"Press ""enter"	appears as	Press "enter

ReGIS also recognizes four control characters within a text string:

Character	Keystroke	Description
C _R	Ctrl-M	Carriage return—horizontally returns the cursor to the position where the string started
L _F	Ctrl-J	Linefeed—moves the cursor down one line
B _S	Ctrl-H	Backspace—moves the cursor back one position; this lets you overstrike a character
H _T	Ctrl-I	Horizontal tab—moves the cursor forward one space, using the current value for text spacing

Select Character Set

The character set option lets you select which character set to use for ReGIS text strings. You can use the built-in character sets shown in “Selecting and Mapping Character Sets” in the *Technical Reference Manual*, or you can load your own (see page 133 for more on the load command). The character sets selected for ReGIS text do not change when you exit or enter ReGIS. The default character sets loaded for ReGIS text are the ASCII character set and the ISO Latin-1 supplemental graphic set.

Character sets are accessed the same way in ReGIS mode as they are in text mode. However, in ReGIS mode, the “in-use” table is distinct and can contain different character sets from the in-use table for text mode. See “Character Set Support” in the *Technical Reference Manual* for more information about how the in-use tables are defined.

You can select a character set for ReGIS text.

Format: T(A<0 to 3>)

A0 selects a set from the built-in character sets, while A1, A2, or A3 selects a set that can be loaded. The built-in set occupies an 8-bit character code table, with the left half (GL) used for 7-bit characters and the right half (GR) for 8-bit characters. By default, the ASCII character set is automatically mapped into GL when you enter ReGIS, and the ISO Latin-1 supplemental graphic set is mapped into GR. The set you can load with A1, A2, or A3 can include only 7-bit characters, and cannot have more than 96 characters.

When selecting the built-in character sets with A0, you can use the following commands to select a character set to map into GL, GR, or both halves of the in-use table. Only 7-bit character sets can be mapped into GL; either 7-bit or 8-bit character sets can be mapped into GR.

```
Format: T(A0(L"<character set designator>"))
        T(A0(R"<character set designator>"))
        T(A0(L"<designator>",R"<designator>"))
```

The following tables list the designators for the 7-bit and 8-bit character sets.

7-Bit Character Set Designators

Character Set	Text Command
ASCII (default)	(B
DEC Special Graphic	(O
DEC Technical	(>

National Replacement Character Sets

British	(A
Dutch	(4
Finnish	(5
French	(R
French Canadian	(9
German	(K
Italian	(Y
Norwegian/Danish	('
Spanish	(Z
Swedish	(7
Swiss	(=
Portuguese	(%6

8-Bit Character Set Designators

DEC supplemental graphic	)%5
ISO Latin-1 supplemental (default)	-A
User-preferred supplemental (94-character set)	)<

In the following example, the DEC Technical character set is selected for GL:

```
T(A0(L"(>"))
```

In this example, the ISO Latin-1 character set is selected for GR:

```
T(A0(R"-A"))
```

In this example, the British character set is selected for GL, and the DEC Technical set is selected for GR:

```
T(A0(L"(A",R"(>"))
```

Character Spacing

You can space between ReGIS text characters in three ways:

- ▲ Select a standard character cell size (explained in the next section). The spacing value for that size is used.
- ▲ Select a character orientation.
- ▲ Specify a relative X and Y value for spacing, using the format below.

To specify a relative X and Y value for spacing between ReGIS characters, use the following command:

Format: T<[X,Y] position>

The X and Y positions are relative values, even if no sign is given. After each character is drawn, the X and Y values tell ReGIS how far to move the cursor before drawing the next character. A negative X spacing value writes text from right to left; a Y spacing value writes text vertically. Changing the character spacing does not change the baseline orientation of characters.

In the following example, the cursor is positioned, a spacing value of 25 pixels is specified, and four letters are written.

```
P[100,100]
```

```
T[25]"ABCD"
```

Character Size

There are three ways to select a character size: select a standard character size; define a display cell size; or define a unit cell size. The *display cell size* is the area that a given character takes up on the screen. This area includes spacing between characters and lines. The *unit cell size* is the size of the character within the display cell.

If you think of the display cell as a box, the unit cell is its contents. Increasing only the display cell size of a character does not increase the size of the character itself; it increases only the area around the character. To increase the size of the characters, you must increase the unit cell size.

The following command selects a standard character cell size:

Format: T(S<0 to 16>)

When you select a standard cell size, the display cell size, unit cell size, and character spacing are already defined. The number 0 to 16 selects a standard size from the table below. Character spacing is expressed as in the previous section.

Text Command: Standard Character Cell Sizes

Standard Size	Display Cell Size	Unit Cell Size	Character Spacing
S0	[9,10]	[8,10]	[9,]
S1 (default)	[9,20]	[8,20]	[9,]
S2	[18,30]	[16,30]	[18,]
S3	[27,45]	[24,45]	[27,]
S4	[36,60]	[32,60]	[36,]
S5	[45,75]	[40,75]	[45,]
S6	[54,90]	[48,90]	[54,]
S7	[63,105]	[56,105]	[63,]
S8	[72,120]	[64,120]	[72,]
S9	[81,135]	[72,135]	[81,]
S10	[90,150]	[80,150]	[90,]
S11	[99,165]	[88,165]	[99,]
S12	[108,180]	[96,180]	[108,]
S13	[117,195]	[104,195]	[117,]
S14	[126,210]	[112,210]	[126,]
S15	[135,225]	[120,225]	[135,]
S16	[144,240]	[128,240]	[144,]

You can also define your own display and unit cell size. The following is the command for defining a display cell size:

Format: T(S[<width,height>])

The following is the command for defining a unit cell size:

Format: T(U[<width,height>])

The unit cell size, which defines the width and height of characters, should be close to the display cell size. Otherwise, the following can occur:

- ▲ If the display cell size is larger than the unit cell size, the excess area around each character is treated as pattern bits. If replace writing or erase writing mode is in effect, the excess area is drawn in the background color; otherwise, the excess area remains unchanged.
- ▲ If the display cell size is smaller than the unit cell size, only the part of the character that can fit in the display cell is displayed.

Height Multiplier

By specifying a height multiplier with the text command, you can change the height of the display and unit cells to the same value without changing their width.

Format: T(H<1 to 256>)

The height for the new character is the value you specify (from 1 to 256 pixels) multiplied by 10. The horizontal character width and the spacing between characters remain unchanged.

For example, the following commands position the cursor, set the character size to the standard size S2 (16 pixel wide by 30 pixel high characters), and write a string of text. Then, the default character height is multiplied by 5, and another string is written. The character width and spacing are not changed.

```
P[50,50]
T(S2)'30 pixels high'
T(H5)'50 pixels high'
```


Size Multiplier

By specifying a size multiplier, you can multiply both the standard character height and width by the same or different factors.

Format: T(M[width<1 to 16>,height>])

This sets the unit cell size to the width you specify multiplied by 8, and the height you specify multiplied by 10.

String Tilt

This option determines a tilt angle for text strings. Each character lies along the same tilted baseline, so that the entire string appears pivoted around the starting point of the string.

Format: T(D<text string angle>,S<0 to 16>)

The tilt angle must be in 45-degree increments. The S<0 to 16> option selects a character cell size from one of the 17 standard cell sizes (see the table on page 126); the cell size is used to determine the spacing between characters in the tilted string. (By itself, the D option indicates only character rotation.)

Because screen pixels are farther apart diagonally than they are horizontally or vertically, characters will appear distorted when writing strings at angles other than 0 or 180 degrees. There is only slight distortion at 90 and 270 degrees, while the distortion is more apparent at 45, 135, 225, and 315 degrees.

You can partially correct for character distortion when writing tilted strings by following this general guideline:

- ▲ Select a character size for the tilted text that has a width about two thirds the width of the untilted text.

For example, if you write a string of untilted text using the standard character size S15, writing a tilted string using standard size S10 will produce characters that look approximately the same size. Size 10 characters are 80 pixels wide, which is two thirds the width of the size 15, 120-pixel wide characters.

Likewise, if you write a string of untilted characters using size 5 characters, which are 40 pixels wide, use a tilted string using size 3 characters, which are 24 pixels wide. The size 3 characters are approximately two-thirds as big as size 5 characters, and look about the same size as the untilted, size 5 characters.

However, even the resized characters may appear distorted, and you may need to make a character height correction. Do this by specifying a height multiplier in addition to a character size. The syntax of the text command becomes:

Format: T(D<text string angle>,S<0 to 16>,H<1 to 256>)

As explained for the height multiplier option on page 127, the value for H is multiplied by 10, the height of the default character cell.

For example, the following command writes a string of text at 45 degrees, using character size 3, and applying a height multiplier of 5. This results in characters that are 24 pixels wide and 50 pixels high.

```
T(D45,S3,H5)
```

String and Character Tilt

A variation on the string tilt option lets you specify both the tilt of the text string and the tilt of each character within the string. By using different string and character tilts, the entire string can be pivoted around a point at the start of the string, with each character pivoted around a point at the start of the character.

Format: T(D<text string angle>,S<0 to 16>,D<char angle>)

Like the string tilt option, character tilts must be at 45-degree increments. If no character tilt is specified, characters are tilted at the same angle as the text string. To offset distortion, you can apply a height multiplication factor. For example:

```
P[50,50]
```

```
T(D315,S3,H5,D180)'This is the text'
```

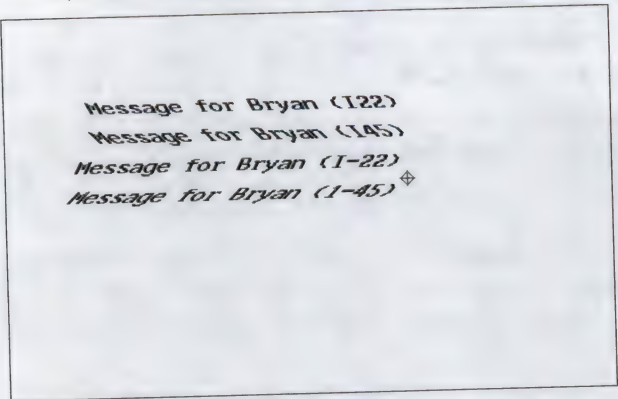

Italics

With the italics option *I*, you can indicate the degree and direction of character slant: a positive number makes characters appear slanted up and to the left, while a negative number makes them appear slanted up and to the right. The angle of the slant can be plus or minus 0 degrees, 22 degrees, or 45 degrees.

Format: T(I<angle>)

The following example shows a line of text written using size S2 at 22 degrees, 45 degrees, minus 22 degrees, and minus 45 degrees. Italics slant is *not* the same as character or string tilt.

```
P[100,100]
T(S2)
T(I22) "Message for Bryan (I22)"
P[100,125]
T(I45) "Message for Bryan (I45)"
P[100,150]
T(I-22) "Message for Bryan (I-22)"
P[100,175]
T(I-45) "Message for Bryan (I-45)"
```



Message for Bryan (I22)
Message for Bryan (I45)
Message for Bryan (I-22)
Message for Bryan (I-45) ♦

Text Command—Italicized Text

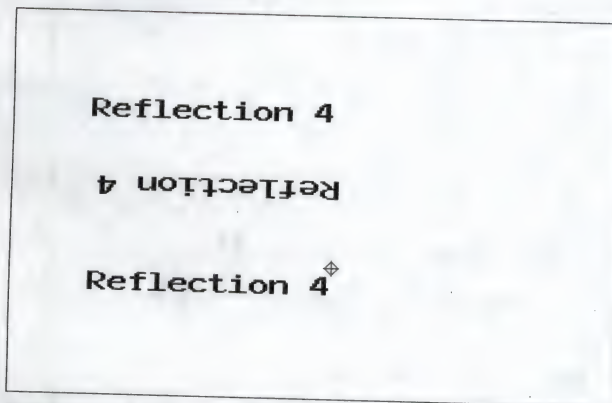
Temporary Text Control

Temporary text control options let you control the characteristics of a limited string of text; when the string ends, text controls are returned to their previous values.

Format: T(B) <options> (E)

The following commands write the phrase "Reflection 4," begin a temporary text control to write the phrase upside down, and then end the temporary control. After the temporary control is ended, the text has the same attributes it did before the temporary control was begun.

```
P[100,100]
T(D0,S3,I0) "Reflection 4"
P[430,250]
T(B) (D180,S3) "Reflection 4" (E)
P[110,330]
T "Reflection 4"
```



Text Command—Temporary Text Control

Temporary Write Control

Temporary write control options provide temporary control over text characteristics such as intensity and the kind of writing: overlay, erase, replace, or complement. See page 103 for details about write command options.

Format: T(W(<suboptions>)<text controls>)

The temporary write controls remain in effect until a new command key letter is used, another temporary write control command is encountered, or the ReGIS command line is resynchronized (;). The temporary write controls apply to the specified text controls only.

Positioning Text with PV Values

You can use the pixel vector system to move text relative to the current baseline. Superscripting, subscripting, and overstriking are all possible. Each PV value moves the character one half of the current display cell size, along the PV direction specified. PV multiplication factors have no effect.

Format: T<PV value>

The following table lists the effect of each of the eight PV values.

PV Value	Character Movement
0	Forward one half character cell
1	Superscript: Up from baseline and to right of the previous character
2	Superscript: Straight up from baseline
3	Up from baseline and toward the previous character, partially overwriting the previous character
4	Overstrike: Backward one half character cell; use 44 to overstrike
5	Down from baseline and toward the previous character, partially overwriting the previous character
6	Subscript: Straight down from baseline
7	Subscript: Down from baseline and to right of the previous character

The following commands write the mnemonic for form feed (F_F) and then return to the baseline:

```
P[300,200]
T(S2) "F"6"F"1
```

Load Command

The text command selects any one of the built-in character sets; the load command, on the other hand, lets you design your own character sets. There are two steps required to load your own character set:

1. Select a number from 1 to 3 for the character set you want to load.
2. Load the character cells (up to 96 characters per set are possible).

Once the character set is created, the select option of the load command is used to load or reload a given set. Although you can select the built-in character set 0, you cannot load it with the select option.

Select Character Set

When you select a character set, all ReGIS load commands apply to this set until you select another set. Other types of ReGIS commands can be used without affecting the selected character set. For example, you can change the appearance of text with the text command options, without affecting the selected character set.

Format: L(A<0 to 3>)

Specify Name

You can specify a name up to ten characters long for the character set you are loading. By giving the character set a name, you can receive a meaningful response when requesting a report about the loaded character set.

Format: L(A"<name>")

You can also give a name to the default character set, which is selected by the text command T(A0). By default, the A0 set has no name. When you use the report command to get the name of a character set, an unnamed set is reported as (A " ").

Load Character Cell

The following command lets you create each character in your selected set. The defined character occupies an 8 pixel wide by 10 pixel high unit cell.

Format: L"<ASCII character>" <hex pairs...>

The <ASCII character> is a single letter or number used to identify the loaded character; it does *not* represent the shape of the loaded character. For example, if you want to create the symbol for copyright, you might call the character "c".

The <hex pairs> define which pixels are on and off within the 8 x 10 character cell. The pairs can be separated by commas for better readability. Each hex code in a hex pair represents four pixels of a cell row: the first hex code of the first hex pair defines the pattern of the left four pixels in the top row of the cell; the second hex code of the first hex pair defines the right four pixels in the first row. Moving from top to bottom through the unit cell, the second hex pair defines the second row of the cell, the third pair defines the third row of the cell, and so on.

The hex codes that create the 16 possible four-pixel patterns are shown in the following table. An example follows the table.

Load Command: Hex Codes

Hex Code	Binary Code	4-Pixel Bit Pattern
0	0000	
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
A	1010	
B	1011	
C	1100	
D	1101	
E	1110	
F	1111	

Blank rows that are above rows with pixels on must be defined; undefined rows at the bottom of the character cell, however, are assumed to be blank.

The following example selects the character set 1, assigns it the name *symbols*, and loads a character with the copyright symbol. Its call letter is *c*.

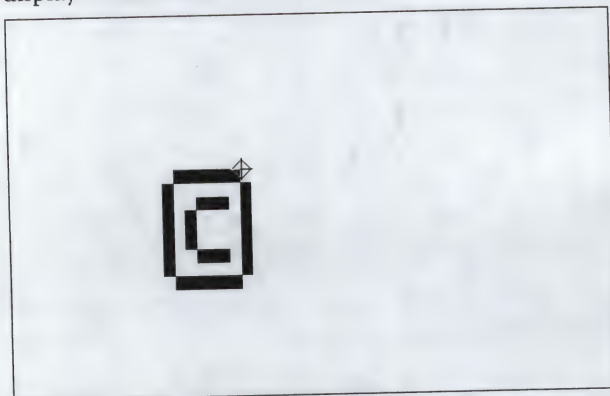
```
L(A1"symbols") "c" 7E,81,9D,A1,A1,A1,9D,81,7E
```

The commas between the hex pairs are not required: they are included for readability only. The next figure shows an enlarged view of the character cell with its corresponding hex pairs.

Character Cell	Hex Code
	7E
	81
	9D
	A1
	A1
	A1
	9D
	81
	7E

Load Command—Designing a Character Cell

The following figure shows what the character looks like when it is displayed on the screen using a standard cell size of S12. The command to select character set 1 and display the character is `T(A1,S12)"c"`.



Load Command—Displaying a Character Cell

Report Command

The report command can perform two different functions:

- ▲ Request information on the current status of ReGIS operations.
- ▲ Enter graphics input mode.

This chapter explains the options that can be used with the report command.

Cursor Position

This option requests the current cursor position; ReGIS reports the absolute screen coordinates:

Format: R(P)

Macrograph Contents and Storage

ReGIS can report on the contents of a specific macrograph with the following command:

Format: R(M(<call letter>))

The <call letter> is the letter that identifies the macrograph. The report that is sent to the host begins with @=<call letter> and ends with @;R. This option is affected by the setting of the Enable Macrograph Reports check box in the ReGIS Setup dialog box. By default, reports are enabled. You may want to disable macrograph reporting for security reasons.

The next command tells ReGIS to report the total amount of space allotted to macrograph storage, and the amount of free space.

Format: R(M(=))

The report sent to the host is in the form *aaaa,tttt* where *aaaa* is the amount of storage still available, and *tttt* is the total amount of storage that was allocated.

Character Set

ReGIS can report the name of the character set currently selected for load command operations.

Format: R(L)

The name of the set is reported in the following format:

(A"<character set name>")

Error

The following option requests a report on the last error detected by the ReGIS command parser; it must be given before the ReGIS command line is resynchronized. The resynchronization character (;) clears all errors.

Format: R(E)

The latest error is reported in the form "<error code>, <ASCII code>". The <error code> is a decimal value followed by a comma. The <ASCII code> is a decimal ASCII code. The following table lists the possible error reports and their meanings.

R Command: Error Conditions

Error condition	Report	Meaning
No error	"0,0"	No error since last synchronization.
Ignore character	"1,<n>"	A character with an ASCII decimal value of <i>n</i> was ignored.
Extra option coordinates	"2,48"	S(H[X,Y] [X,Y]) contains more than two coordinates.
Extra coordinate values	"3,48"	[X,Y] contains more than two coordinates.
Alphabet out of range	"4,48"	L(A<0 to 3>) contains a number outside of the <0 to 3> range.

Error condition	Report	Meaning
Begin/start overflow	"7,66" or "7,83"	The limit of 16 (B) and (S) options for position and vector commands has been exceeded.
Begin/start underflow	"8,69"	A position or vector command contains an (E) option without a matching (B) option.
Text standard size error	"9,48"	A text command has selected a standard character size outside of the <0 to 16> range.

Graphics Input Modes

Graphics input modes let you position the cursor and send a position report. The report is sent in response to the interactive position report command described in the next section.

Input mode can be either *one-shot* (the default) or *multiple*. In one-shot mode, the application running on the host is suspended until ReGIS sends a position report. In multiple input mode, a number of cursor position reports can be sent without leaving graphics input mode.

The following command indicates that the interactive position report should use one-shot mode:

Format: R(I0)

The input cursor appears on the screen, and you use the mouse or arrow keys to position it. You remain in one-shot mode until a cursor position request is received; a position report is sent when you press the mouse button or an active non-arrow key. Once the position report is sent, you exit one-shot mode.

The following command indicates that the interactive position report should use multiple input mode:

Format: R(I1)

The input cursor appears, and you use the mouse to position it. Multiple mode allows more than one cursor position report to be sent without exiting input mode. In this mode, cursor position reports are sent whenever you press the mouse button, or when an interactive position report command is received.

To exit multiple input mode, use the command that indicates one-shot mode:

R(I0)

Interactive Position of Input Cursor

With this option, an application can request an input cursor position report in either one-shot or multiple graphics input mode. You use the mouse to position the cursor.

Format: R(P(I))

In one-shot mode, ReGIS sends the position report when you press an active non-arrow key or the mouse button. In one-shot mode, you can use the mouse or arrow keys to position the input cursor; for a list of modifier keys you can use with the arrow keys to accelerate the input cursor, see page 140.

In multiple mode, ReGIS sends a position report immediately when you press the mouse button. You cannot use the arrow keys in multiple input mode to position the cursor.

Cursor Movement

If you use a locator device, such as a mouse or graphics tablet, the input cursor moves when you move your device. When the graphics input cursor appears on the screen, it can be moved with the locator device or with the arrow keys on the numeric keypad. The **Ins** key on the keypad makes the cursor move twice as fast, while the **Shift** key moves the cursor ten times as fast. Using both **Ins** and **Shift** at the same time moves the input cursor twenty pixels at a time.

Keystroke

↑ ↓ ← →

Ins-↑ ↓ ← →

Shift-↑ ↓ ← →

Shift-**Ins**-↑ ↓ ← →

Effect

Moves cursor 1 pixel at a time

Moves cursor 2 pixels at a time

Moves cursor 10 pixels at a time

Moves cursor 20 pixels at a time

Position Report Format

In one-shot mode, a position report is sent only after the interactive cursor position command is received by ReGIS. The report contains the code for the key pressed (or a mouse button code from the table below), followed by the cursor's absolute position and a carriage return.

For example, a position report of `a[225,134]` means that when the interactive cursor position command was sent, the user placed the input cursor at coordinates [225,134] and pressed the `[a]` key. A position report of `csI241~[350,75]` means the user placed the input cursor at [350,75] and pressed the mouse button.

In multiple input mode, a cursor position is sent whenever one is requested by the host application or the mouse button is pressed. If an interactive position was *not* requested, the format of the position report is the same as above for one-shot input mode. If an interactive position *was* requested, the format of the position report is `csI240~[X,Y]`. The `csI240~` is a "null button" sequence; it indicates that the mouse button was pressed in response to an interactive position request.

Button	Code (when pressed)
1 (left)	<code>^cs_I241~</code>
2 (middle)	<code>^cs_I243~</code>
3 (right)	<code>^cs_I245~</code>
4	<code>^cs_I247~</code>

The left, middle, and right designations in this table refer to the buttons on Digital's three-button mouse. By default, the left PC mouse button is configured to correspond to the left button on a Digital mouse (see page 71 for more information).

The response that the mouse button sends when the button is pressed and released can be defined using the DECLBD function; see the *Technical Reference*.

Macrograph Command

A macrograph, as its name implies, is a macro that stores ReGIS graphics commands. Instead of redrawing the same image each time you need it, you can store the ReGIS commands for the image under a macrograph call letter, then replay the macrograph each time you want the image. You can store up to 26 macrographs. You can also nest macrographs within other macrographs, up to 16 levels deep. By default, 10k of memory is allocated for storing VT420 macros and ReGIS macrographs (the field name is Macro Storage in the General Setup dialog box).

Define Macrograph

A macrograph is identified by a single letter that is not case sensitive.

Format: @:<call letter> <definition>@;

Macrograph commands are not drawn as the macrograph is defined. You must invoke the macrograph to draw the image.

The following example defines a macrograph, identified as S, to draw a square relative to the cursor position. Each side of the square is 100 pixels long.

```
@:S V[+100] [, +100] [-100] [,-100]@;
```

Invoke Macrograph

The following command invokes a macrograph:

Format: @<call letter>

This command invokes the macrograph defined above:

```
@S
```

Clear Single Macrograph

The syntax for deleting a single macrograph is as follows:

Format: @:<call letter>@;

Clear All Macrographs

To clear all defined macrographs, use the following command (the period is required):

Format: @.

Tektronix 4014 Graphics

Tektronix 4014 is an additional terminal emulation that is included in Reflection 4.

Chapter 18, *Using Tektronix Graphics*, offers detailed information on Tektronix operating modes, keystrokes, and commands. It also provides a sample graphics session.

Chapter 18

Using Tektronix Graphics	147
Starting Reflection with the Tektronix Option	147
4014 Terminal Emulation Uses	147
4014 Mode Characteristics	147
Sample Graphics Session	149
Tektronix Keystroke Summary	150
Printing the Graphics Screen	152
Obtaining Terminal Status Information	153
ASCII Control Codes	154
Determining Graphics Plot Coordinates	155
Keyboard Mapping Notes	156

Using Tektronix Graphics

Reflection emulates the Tektronix 4014 terminal in essentially all aspects, so you interact with graphics programs exactly as they are documented for the terminal.

The *Technical Reference* explains how to set up Reflection for Tektronix graphics emulation and describes the control functions associated with the Tektronix terminal, including raster writing mode.

Starting Reflection with the Tektronix Option

Type `R4` at the DOS prompt, then use `[Alt]-[G]` to toggle into Tektronix mode. The screen appears blank; the labels at the bottom of the screen disappear and the cursor is in the upper left corner.

The Tektronix screen uses bit-mapped graphics to display both alphanumeric characters and graphics. In most cases the application software you are running automatically switches between the VT and Tektronix emulations.

4014 Terminal Emulation Uses

Reflection's 4014 terminal emulation lets you run host computer graphics programs with your PC. Each application program is different, so review the program's documentation for specific information.

The host controls all of the graphics features discussed here. You do not need to enter control functions or, in some cases, even turn on Tektronix emulation mode. A sample session is provided to acquaint you with the various features of the Tektronix emulation.

4014 Mode Characteristics

The Tektronix terminal has three primary modes: alphanumeric, graphics display, and graphics input.

Alphanumeric mode

This is the initial state of the 4014 emulation. In this mode, the terminal displays only alphanumeric characters. The position of the alpha cursor shows the next character position. This mode displays 35 rows, with 74 characters per row.

Two margins are active in this mode: margin 0 and margin 1. Margin 0 is flush with the left margin of the screen. When the right margin is reached, a carriage return and linefeed are generated automatically. When the 35th line of input is reached, the cursor is repositioned at the top row halfway across the screen. This is the margin 1 position. This feature allows the user to view two columns of text. When entering or editing lines in margin 0, it is possible to overwrite characters in margin 1.

Graphics display mode

Graphics display mode consists of 1024 by 1024 addressable points. Only 780 of these points can be seen on the vertical axis; 0,0 is the lower left corner of the screen.

The host sends down an ASCII GS (G_S) character to put the terminal in graphics plot mode. A U_S , $ESC F_F$ or C_R returns the terminal to alphanumeric mode, as well as the PAGE (Alt-P) and RESET (Alt-R) function. See page 150.

Once the terminal has been placed in graphics display mode, characters are interpreted by the terminal as vector endpoints. After the first endpoint, which is considered a *dark* vector point, lines are drawn between endpoints.

Graphics input mode (GIN)

Graphics input mode allows the host computer to request the alpha cursor position, the graphics *beam* position (the last endpoint), or the crosshair cursor position, as well as terminal status information. See page 153.

Sample Graphics Session

The graphics features of Reflection's Tektronix emulation are primarily for using graphics programs running on a host computer. However, you can perform some graphics operations independent of a host application.

Follow the steps below to enter and exit graphics mode. An image is created on the screen by sending control codes to the PC with Reflection's command language:

1. Type `R4` at the DOS prompt.
2. Press **Alt-F10** to display the Reflection command line, type `DIR DEMO.PIC`, and press **Enter**. If the file is not listed, copy it into the current directory from the product disk.
3. Press **Alt-G** to toggle to Tektronix emulation. The menu bar disappears, and the cursor is located in the upper left corner of the screen. The PC's video card is in graphics mode, which means that the alphanumeric characters are displayed in bit-mapped graphics.
4. If you have an active host session, type some characters to see how they are displayed on the screen. Otherwise, press **Alt-L** to toggle to local mode, and then type some characters.
5. Press **Alt-P** to clear the screen.
6. Press **Alt-G** to return to the VT emulation and text mode.
7. Choose Graphics from the Setup menu, and then choose Tektronix from the Graphics cascading menu.
8. In the Tektronix Setup dialog box, select the Virtual Screen check box, then choose OK to activate this value.
9. Press **Alt-G** to toggle back to Tektronix emulation, then display Reflection's command line by pressing **Alt-F10** or by choosing Command Line from the Tools menu.
10. Enter `TYPE DEMO.PIC` and press **Enter**. The Tektronix graphics display is turned on and a graphic is displayed on the screen.
11. Press **Esc** to exit Reflection's command line.

Now, practice controlling the display and the cursor:

1. Press **(Alt)-(Z)** to zoom the image. Because the larger Tektronix screen does not fit on the PC's screen, only a portion of the image is shown. Pressing **(Alt)-(Z)** toggles the image between scaled and unscaled. Move the cursor around on the unscaled image by using **(Ctrl)** or **(Alt)** and the arrow keys.
2. Press **(Esc)** and then type **(Ctrl)-(Z)** to turn on the crosshair cursor. You see a fine hairline in the upper left corner of the screen. This is a portion of the crosshair cursor. If nothing appears, press **(Alt)-(L)** to ensure that you are in local mode and try again. When you run graphics applications, the host turns the crosshair cursor on and off.
3. Hold down **(2)**(keypad), and then **(6)**(keypad). The crosshair cursor moves toward the center of the screen. Then hold **(Ins)** while moving the cursor and notice how movement varies.
4. Practice moving the crosshair cursor around the screen, using the arrow keys. Position it at the left edge of the screen, near the vertical center.
5. With the graphics image visible, print the screen with the **(Alt)-(PrtSc)** keystroke.

Tektronix Keystroke Summary

Function	Keystroke and Description
Break	(Ctrl)-(Break) Sends a BREAK signal to the host.
Enter/Exit Tek	(Alt)-(G) When you start Reflection and toggle into the Tektronix emulation, you enter scaled alphanumeric mode. Subsequent use of this keystroke toggles between graphics and text modes.
Erase display	(Alt)-(P) Erases display.
Local/Online	(Alt)-(L) Toggles between local mode and remote mode, which is the same as using the Online field in the General Setup dialog box.

Move cursor Arrow keys:    

Moves the crosshair cursor if it is showing. Holding down two arrow keys allows you to move the cursor diagonally. If the crosshair cursor hits a border while you are viewing a window, the window is repositioned such that the crosshair cursor is centered in the window. If the window hits a border of the Tektronix screen, then just the crosshair cursor moves.

Move window -arrow keys or -arrow keys

Moves the window you are viewing.

Page -P

The Tektronix terminal PAGE CLEAR function.

Print screen -PrtSc

Sends the screen contents to a configured printer. If the Auto Form Feed check box is selected in the Printer Setup dialog box, a form feed is also performed.

Command line -F10

Displays the Reflection command line.

Reset -R

The Tektronix terminal RESET function:

- ▲ Homes the cursor
- ▲ Cancels any print operation in progress
- ▲ Clears the keyboard buffer
- ▲ Initializes the serial communication ports to their last activated values
- ▲ Clears the receive buffer
- ▲ Resets the terminal to its initial status

This function does not erase the screen.

Scaled/Unscaled **[Alt]-[Z]**

Scaled/Unscaled toggle.

If you configure a buffer for the 1024 × 780 Tektronix screen by selecting the Virtual Screen check box in the Tektronix Setup dialog box, this keystroke toggles between the scaled image and a window on the unscaled image. The window center is a crosshair cursor (if it has been turned on).

Set alpha mode **[Alt]-[P]**

Resets to alphanumeric mode and home position. Resets to margin 0 and cancels bypass condition. (See page 153 for a description of the bypass condition.)

Speed cursor **[Ins]**-arrow keys

Holding down **[Ins]** while using the arrow keys moves the crosshair cursor more quickly.

Printing the Graphics Screen

Press **[Alt]-[PrtSc]** to send the contents of the graphics display to the printer. If there is no print buffer configured, the printer should immediately begin printing. If there is a buffer, expect a 3 to 20-second delay (depending on the speed of your computer) while Reflection fills it.

Because printers take a few minutes to print a graphics screen, and your keyboard is locked during this time, a print buffer is recommended for graphics printing. With an adequate buffer, the keyboard is only locked for as long as it takes to fill it. A 32K buffer holds a graphics screen. A form feed occurs if the Auto Form Feed check box is selected in the Printer Setup dialog box (the default).

If you are having problems with graphics printing, be sure that Reflection is configured for your printer and that your printer supports graphics printing. Reflection does not support graphics printing for the printer selections TOSHIBA and OTHER.

Be aware that some printers can use codes other than their native graphics language. For instance, some Okidata Microline 182 printers use Okidata control codes and some use IBM/Epson. Always set Reflection for the control codes your printer uses.

Obtaining Terminal Status Information

When the host computer sends an ESCENQ sequence to the terminal, Reflection responds in the following manner:

▲ Alphanumeric mode

One byte of terminal status information, followed by the four-byte address of the lower left corner of the alpha cursor, followed by the user-configured GIN terminator.

▲ Graphics plot mode

One byte of terminal status information, followed by the four-byte address of the graphics mode beam position, followed by the user-configured GIN terminator.

▲ GIN mode

Four-byte address of the crosshair cursor position, followed by the user-configured GIN terminator.

The terminal status byte that Reflection sends contains the following (bit 7 is the high-order bit):

Bit 7	Parity bit
Bit 6	Always 0
Bit 5	Always 1
Bit 4	0 indicates a configured printer
Bit 3	Always 0
Bit 2	0 if terminal is in graphics display mode, 1 if otherwise
Bit 1	1 if margin 1 is set (i.e., alpha cursor is in the right half of the screen)
Bit 0	Always 1

Any one of the above modes indicates that the terminal is in *bypass* condition. Bypass prevents the terminal from processing or displaying any GIN mode data sent to or echoed from the host.

If the GIN terminator contains a CR character, bypass is cleared when CR is echoed from the host.

If the GIN terminator does not contain a CR, then bypass must be cleared by the host sending (or echoing) any of the following ASCII characters:

BEL	B_{E_L}	Ctrl - G
CR	C_R	Enter ↵
ESC ETB	$E_{S_C E_{T_B}}$	Ctrl - W
ESC FF	$E_{S_C F_F}$	Ctrl - L
LF	L_F	Ctrl - J
US	U_S	Ctrl - Shift - -
SI	S_I	Ctrl - O

ASCII Control Codes

- B_{E_L} Rings the bell and clears the bypass condition.
- B_S Moves the cursor left one position.
- C_R Performs a carriage return, sets the terminal to alphanumeric mode, cancels the crosshair cursor, clears the bypass condition, and performs a linefeed if the CR Effect field in the Tektronix Setup dialog box is set to CR-LF.
- G_S Sets the terminal to graphics mode, and begins the first dark vector point.
- H_T Moves the cursor to the right one character. Also performed with **Ctrl**-**I** or **Tab** from the keyboard.
- L_F Moves the cursor down one line, clears the bypass condition, and performs a carriage return if the LF Effect field in the Tektronix Setup dialog box is set to LF-CR.
- S_I Clears the bypass condition.
- U_S Sets the terminal to alphanumeric mode and clears the bypass condition.
- V_T Moves the cursor up one row.

Determining Graphics Plot Coordinates

Graphics plot data is sent to the terminal as a series of vector endpoints. Each vector is defined by sending its endpoints as a sequence of four ASCII characters. This four-character sequence consists of the high and low-order portions of the Y coordinate, and the high and low-order portions of the X coordinate, in that order.

Each character (byte) contains 2 tag bits plus 5 data bits for either the high-order or low-order bits of the number. After the initial four bytes have been sent to the terminal to establish an endpoint, additional bytes that do not change may not have to be sent again. Low X bytes, however, must always be sent, and low Y bytes must be sent if the high X byte has changed.

For example, to send the endpoint (X,Y) where X = 212 and Y = 489, use:

212 = D4 (Hex), 489 = 1E9 (Hex)

The terminal receives the endpoint as a sequence of 4 bytes in the order high Y, low Y, high X, low X.

The Y coordinate is as follows:

1E9h	= 111101001
	= 0111101001 (make it 10 bits long)
High Y byte	= tag bits (01), 5 high-order bits of Y value
	= 01 01111
	= 2Fh
	= the / character is sent to the terminal
Low Y byte	= tag bits (11), 5 low-order bits of Y value
	= 11 01001
	= 69h
	= the i character is sent to the terminal

The X coordinate is as follows:

D4h = 11010100
 = 0011010100 (make it 10 bits long)

High X byte = tag bits (01), 5 high-order bits of X value
 = 01 00110
 = 26h
 = the & character is sent to the terminal

Low X byte = tag bits (10), 5 low-order bits of X value
 = 10 10100
 = 54h
 = the T character is sent to the terminal

Keyboard Mapping Notes

You can remap any Tektronix function to another keystroke—see page 40 in “Keyboard Mapping.” The key names used to remap the break, print screen, and command line functions are also discussed there.

Application Program Interface

Reflection's Application Program Interface (API) provides a way for a foreground program to "call" a background copy of Reflection so that it can initiate file transfers, log on to a host computer, dial a modem, or perform other communications functions. The foreground API program controls a background copy of Reflection, and makes it do anything that a user normally does:

- ▲ Command language
- ▲ Keystroke entry
- ▲ File transfers
- ▲ Screen display
- ▲ Popping Reflection to the foreground

Application Program Interface support is included in all Reflection PC products for HP and DEC emulation. With version 5.0, all Reflection PC products also include the libraries for C, Pascal, BASIC, and Professional BASIC required for API program development.

Chapter 19

API Support Files	161
------------------------------------	------------

Chapter 20

Summary of API Function Calls	165
Status Calls	165
Command Language	165
Keyboard Support	166
Screen Support	166
Datacomm Support	167
Session Support	167
Miscellaneous	167

Chapter 21

API Function Calls	169
HP User Key Label Structure (Reflection 1 and Reflection 7)	188
DEC Softkey Structure (Reflection 2 and Reflection 4)	189
DEC UDK Structure (Reflection 2 and Reflection 4)	189
Structure Definition	192
Identifying Numbers	193
0 and 1 Settings	194
Structure Definition	198

Chapter 22

API Error Codes	259
----------------------------------	------------

Chapter 23

How to Write an API Application	261
Queued vs. Synchronous Commands	261
Synchronous Commands	261
Asynchronous Commands	262
Queued File Transfer	263
Combining Queued and Synchronous Commands	263
Timeouts	264
Keyboard Reads	265
Conflicting Commands	266
Preventing User Exits	266

Uninstalling Reflection	267
Datacomm in api_docommand Sequences	267
Configuration Issues	268
Compiling and Linking with API Libraries	269
Building API Applications with Borland Turbo Pascal	269
Building API Applications with Microsoft QuickBASIC	269
Building API Applications with Microsoft Professional BASIC	270
Building API Applications with Microsoft C	271
Building API Applications with Borland C	271

Chapter 24

Converting Command Language to API	273
Command Language Dialing Program	274
C Version	276
Queuing Keystrokes	279

API Support Files

To load Reflection with Application Program Interface (API) support, use the /W startup switch. All Reflection PC products for HP and DEC emulation include API support, but the startup switch is required to activate it. Using this switch adds about 4000 bytes to Reflection.

Reflection's API provides hooks for API programs written in assembler, C, Borland's Turbo Pascal, Microsoft's Quick BASIC, and Professional BASIC.

With version 5.0, all Reflection PC products also include development libraries for C, Pascal, BASIC, and Professional BASIC: the libraries are on a support disk labeled *Application Program Interface*. The API disk also includes a full description and examples of each API function call.

The following is an overview of the support files and demo programs that are included on the API disk:

File or Directory Name	Description
API.DOC	Contains a full description of each API function call with examples.
<APIDEMO> Directory	
Utility programs that use the API feature	
HPCMD.EXE	Run HP commands at the DOS prompt and see the results displayed on PC screen.
HPCMD.C	Source for HPCMD.EXE.
DOAPI.EXE	Send a command to Reflection in background to transfer a file or run a command script.
DOAPI.C	Source for DOAPI.EXE.
\$.EXE	Run VAX commands from the DOS prompt with results displayed on PC screen.
HP.BAT	Sample DOS batch file that demonstrates how HPCMD.EXE can be used to check for a file on the HP 3000 from DOS.
APIDEMO.DOC	Documentation on the above files.

<Pascal> Directory

APIUNIT.PAS

APIUNIT.TPU

P_APILIB.OBJ

SAMPLE.PAS

SAMPLE.EXE

<BASIC> Directory

API.INC

SAMPLE.BAS

SAMPLE.EXE

B_APILIB.QLB

B_APILIB.LIB

BAS_API.DOC

PBAPILIB.LIB

<C> Directory

API.H

SAMPLE.C

SAMPLE.EXE

C_SAPI.LIB

Turbo Pascal library support for API

Source of API UNIT file.

TPU Unit file, created by compiling APIUNIT.PAS. Code can be included into your Pascal program via the 'uses' statement.

API library in OBJ format. Used to create new APIUNIT.TPU. You may customize APIUNIT.PAS as needed and recompile via the TPC command.

Sample Pascal program which does API calls.

Compiled version of SAMPLE.PAS.

BASIC library support for API*

Include file listing all function declarations. Use the \$INCLUDE metaccommand to include this file in your BASIC source.

Sample BASIC program using API calls.

Compiled version of SAMPLE.BAS.

Microsoft BASIC Library with API calls. Use with Microsoft BASIC environment.

Standard API LIB for use with command line BC compiler and linker.

List of API function declarations.

API library for Professional BASIC.

C library support for API

Include file with API structure definitions.

Sample program that uses API calls.

Compiled version of SAMPLE.C.

Small model API library.

* Microsoft's Quick BASIC and Professional BASIC are both referred to in this manual simply as "BASIC." The only difference between the two is that Professional BASIC must be linked with the PBAPILIB.LIB API library.

C_CAPI.LIB	Compact model API library.
C_MAPI.LIB	Medium model API library.
C_LAPI.LIB	Large model API library.

Summary of API Function Calls

The API function calls are summarized here. The Reflection support disk labeled *Application Program Interface* contains a full library with examples of each API function call. A total of 34 function calls are available in this release of Reflection. A full description of each function call with examples follows and is also provided on your *Application Program Interface* disk. Currently C, Turbo Pascal, and Microsoft Quick BASIC and Professional BASIC libraries (on disk) are available for linking with API applications.

Status Calls

Determine Reflection and API status.

Function call	Description	Page
api_getinfo	Get static information	192
api_getstatus	Get dynamic information	198
api_instchk	See if API present	206
api_rcheck	See if Reflection installed	223
api_setgetintstate	Enable break out from current command	244

Command Language

Synchronous—commands that must complete before control is returned to the API application.

Function call	Description	Page
api_docommand	Do a command	182
api_endcommands	End a command sequence	185
api_found	Return value of the FOUND Boolean	186
api_getvar	Return a command language variable	202
api_setvar	Set a command language variable	247
api_startcommands	Start a command sequence	250

Asynchronous—commands that are queued by Reflection so that the API application does not have to wait for completion.

Function call	Description	Page
<code>api_clrcmdq</code>	Flush the queue of commands	176
<code>api_cmdstatus</code>	How much free space is in the command queue	180
<code>api_qcmd</code>	Queue a new command	218

Keyboard Support

Queue keystrokes.

Function call	Description	Page
<code>api_clrkeybd</code>	Flush the keyboard queue	178
<code>api_getfkey</code>	Get a function key	188
<code>api_keystatus</code>	How much free space is in the keyboard queue	209
<code>api_qkeys</code>	Queue some keystrokes	220
<code>api_setfkey</code>	Set a function key	242

Screen Support

Read and search Reflection's screen.

Function call	Description	Page
<code>api_atrbscreen</code>	Read screen with character attributes	171
<code>api_screenread</code>	Read the text screen (minus function keys)	234
<code>api_searchscreen</code>	Search the screen for a string	238

Datacomm Support

Read and write to datacomm from the API application.

Function call	Description	Page
api_assertdc	Re-assert control over the datacomm port	170
api_rdchar	Read a character from com port*	226
api_releasedc	Release the datacomm port	230
api_writeasync	Write character to com port*	255
api_xmitstatus	Check status of transmit buffer	257

Session Support

Start, control, and end API sessions, and pop Reflection into the foreground.

Function call	Description	Page
api_block	Give Reflection some CPU time	173
api_ndcpopup	Pop up Reflection and buffer incoming data	211
api_popup	Pop up Reflection	216
api_wait	Wait until Reflection is free	253

Miscellaneous

Create buffers and reset Reflection.

Function call	Description	Page
api_cancelbuf	Cancel use of buffer	175
api_offerbuf	Provide buffers for queuing keys or commands	213
api_reset	Do a hard reset of Reflection	232

* Serial or network connection.

API Function Calls

This chapter consists of a description of each of the API function calls with short examples for assembler, C, Pascal and BASIC. The Pascal interface requires version 5.0 of Turbo Pascal or greater. The BASIC interface requires Microsoft QuickBASIC version 4.50 or greater, or Professional BASIC.

All API functions return zero on successful completion. On error, a non-zero value is returned. See "API Error Codes" on page 259 for a list of all return values.

The following table shows the assembly language registers used by Reflection's Application Program Interface.

Assembly Language Interface

AX	=	2A52h
BP	=	ABCDh
DX	=	Ø
CH	=	Ø (reserved for future use)
CL	=	API function code
ES		Used to pass parameters
BX		Used to pass parameters
SI		Used to pass parameters
DI		Used to pass parameters
INT		21h (use MS-DOS Interrupt 21h to access API)

Versions of Reflection prior to 4.2 used a slightly different calling convention (AH was ØDE52h), which still works. The new convention, however, is more reliable and should be used on new applications.

api_assertdc

Tells Reflection to take the serial hardware away from a foreground task. Reflection re-initializes datacomm back to its normal baud rate, parity, etc. Reflection will continually re-take the hardware from another program if necessary.

Input: CX = 19
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0 Should not return an error

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,19
int 21h
```

BASIC, C, Pascal

For examples, see `api_releasedc` on page 230.

api_atrbscreen

Read text and color attributes from Reflection's background screen. Read can extend to function key display. Color attributes are IBM PC attribute bytes. Client program may read Reflection screen and re-display with same color attributes.

Input: CX = 31
 AX = 2A52h
 BP = ABCDh
 DX = 0
 BL = ROW 0-relative
 BH = COL 0-relative
 ES:DI Points to buffer to contain screen text
 SI = Number of characters to read
 Each character is 2 bytes with this function. For each character, the first byte is the character from the IBM character set and the second byte is the attribute that determines how the byte is displayed.

Output: AX = 0 No error
 AX = 103h Error; starting read location is off screen
 AX = 106h Error; screen in graphics mode

Assembler and C

See example for api_screenread on page 234.

BASIC

Example—Read the Reflection screen and print characters with correct color attributes:

```
' $INCLUDE: 'api.inc'
'
' Allocate space for reading 80 columns by 25 rows by
' two bytes per character.
x$ = space$( 80 * 25 * 2 )
```

```

' Ask for 80*25 characters starting at row 0, column 0
i% = api_atrbscreen%( x$, 0, 0, 80 * 25 )
'
' Make default segment the segment of the video display
' (B000 for monochrome)
DEF SEG = &HB000
FOR i = 1 TO 80 * 25 * 2
POKE i-1, ASC( MID$( x$, i, 1 ) )
NEXT i
end

```

Turbo Pascal

function api_atrbscreen(length,col,row:integer;var x:buffer):integer;

Also see the example for api_screenread on page 234.

Example—Read 100 characters off of screen and put ASCII bytes only into string variable. Read starting at row 10 in column 1:

```

program ReadScrnAttr ( input,output );
{$V-}
uses   apiunit;
var    scrn : buffer;
        row : integer;
        col : integer;
        i : integer;
        scrtext : string[100];
begin
    row := 10;
    col := 1;
    i := api_atrbscreen ( 100, col, row, scrn );
    i := 0;
    while i < 100 do
        begin
            scrtext[i+1] := scrn[ i*2 ];
            i := i+1;
        end;
    scrtext[0] := chr( 100 );
    writeln( scrtext );
end.

```


api_block

Give Reflection some CPU time. This should be done if the API client program is waiting for Reflection to complete some queued commands or keystrokes, but wants to maintain CPU control (`api_block` returns immediately).

Input: CX = 27
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0 Command or key queues have been completed
 AX <> 0 Reflection is busy processing queued commands
 or keystrokes

Assembler

Example—Give Reflection CPU time while waiting for user input:

```
idle_loop:
    mov  ah,1           ; Check keyboard status
    int  16h           ; BIOS keyboard interrupt
    jnz  key_waiting    ; Exit if user has typed key
    mov  ax,02A52h
    mov  bp,0ABCDh
    xor  dx,dx
    mov  cx,27          ; Do api_block
    int  21h
    or   ax,ax          ; Is Reflection finished?
    jnz  idle_loop      ; No, see if there is keyboard input
```

C

```
int api_block( void );
```

Example:

```
#include "api.h"
main()
{
    /* Queue two commands and block until complete */

    int status;
    api_qcmd( "wait 0:0:3" );
    api_qcmd( "display '^g'" );
    while ( ( status = api_block() ) != 0 )

        /* If key has been pressed stop waiting */

        if ( kbhit() )
            break;
    return status;
}
```

BASIC

```
DECLARE FUNCTION api.block% CDECL ( )
```

Example—Give Reflection CPU time until file transfer completes or user presses a key:

```
' $INCLUDE: 'api.inc'
'
i% = api.qcmd( "send bigfile to xyz" )
DO
LOOP WHILE inkey$="" and api.block <> 0
```

Turbo Pascal

```
function api_block :integer;
```

For an example, see `api_offerbuf` on page 213.

api_cancelbuf

This function makes Reflection use default keyboard and command buffers. The original buffer is reclaimed by the API client program. Issue this call prior to terminating.

Input: CX = 25
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0 Should not return an error

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,25
int 21h
```

C

```
int api_cancelbuf( void );
```

BASIC

```
DECLARE FUNCTION api.cancelbuf% CDECL ( )
```

For an example, see `api_offerbuf` on page 213.

Turbo Pascal

```
function api_cancelbuf :integer;
```

For an example, see `api_offerbuf` on page 213.

api_clrmdq

Clears the command queue.

Input: CX = 17
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0 Should not return an error

Assembler

Example:

```
mov ax, 02A52h
mov bp, 0ABCDh
xor dx, dx
mov cx, 17
int 21h
```

C

```
int api_clrmdq( void );
```

Example:

```
#include "api.h"
main()
{
    /* Clear command queue and report status */

    int status;

    printf( "Clearing command queue.\n" );
    status = api_clrmdq();
    if ( status )
        printf( "Could not clear queue. Error code: %d\n", status );
    return status;
}
```

BASIC

DECLARE FUNCTION api.clrcmdq% CDECL ()

Example—Clear the command queue:

```
' $INCLUDE: 'api.inc'  
,  
i% = api.clrcmdq
```

Turbo Pascal

function api_clrcmdq :integer;

Example—Clear the command queue:

```
uses    apiunit;  
var     i : integer;  
begin  
    i := api_clrcmdq;  
end.
```

api_clrkeybd

Clear any remaining unprocessed keys out of the keyboard buffer.

Input: CX = 14
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0 Should not return an error

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,14
int 21h
```

C

```
int api_clrkeybd( void );
```

Example:

```
#include "api.h"
main()
{
    /* Try to queue some keys, and clear the queue if this fails */

    int    status;
    char   *keybuffer = "'read mail' UNKNOWN";

    if ( strlen( keybuffer ) <= api_keystatus() )
        status = api_qkeys( keybuffer );
    else {
        printf( "No room in Reflection key queue.\n" );
        return QueueFull;
    }
}
```



```
if ( status == BadKey ) {
    printf( "Unrecognized key name.\n" );
    printf( "Clearing keyboard queue.\n" );
    api_clrkeybd();
}
else if ( status ) {
    printf( "Could not queue keys. Reason unknown.\n" );
}
else
    printf( "Key has been queued.\n" );
return status;
}
```

BASIC

DECLARE FUNCTION api.clrkeybd% CDECL ()

Example—Clear the keyboard buffer:

```
' $INCLUDE: 'api.inc'
'
i% = api.clrkeybd
```

Turbo Pascal

function api_clrkeybd :integer;

Example:

```
uses    apiunit;
var     i : integer;
begin
    i := api_clrkeybd
end.
```

api_cmdstatus

Returns amount of free space available in command queue buffer.

Input: CX = 15
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = *<Amount of free space in bytes>*

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,15
int 21h
or ax,ax
jz No_free_space
```

C

```
int api_cmdstatus( void );
```

Example:

```
#include "api.h"
main()
{
    /* Queue a command if there is room in the buffer */

    char *send_command = "SEND ABC TO XYZ ASCII DELETE";
    if ( api_cmdstatus() >= strlen( send_command ) ) {
        api_qcmd( send_command );
        printf( "Command queued\n" );
    }
}
```

```

    else {
        printf( "Queue full\n" );
        return 1;
    }
    return 0;
}

```

BASIC

DECLARE FUNCTION api.cmdstatus% CDECL ()

Example—Queue a command if there is room in the buffer:

```

' $INCLUDE: 'api.inc'
'
cm$= "SEND ABC TO XYZ ASCII DELETE";
'
' If there is room, queue the command
if ( api.cmdstatus > len( cm$ ) ) then
    api.qcmd( cm$ )
    print "Command Queued"
else
    print "Queue full"
end if

```

Turbo Pascal

function api_cmdstatus :integer;

Example:

```

uses   apiunit;
var    i : integer;
begin
    i := api_cmdstatus;
    writeln ( 'bytes free in command buffer = ', i );
end.

```


api_docommand

Perform a Reflection command synchronously. The call does not return to the caller until the command has been completed or an error is encountered.

During the wait for a Reflection command to complete, you cannot use the hot-key. Reflection will just beep at you. ALERT messages will not blink on and off as normal. The calling program will be frozen until the command completes. Make sure that all commands issued will either time out after a while, or use the `api_setgetintstate()` function, otherwise your API program can crash waiting for a condition that may never be met. A discussion of queued versus synchronous commands begins on page 261; using both types of commands is discussed on page 263.

Use: `WAIT 0:0:10 FOR "^Q"`

Don't use: `WAIT FOR "^Q"`

Input: `CX = 6`
 `AX = 2A52h`
 `BP = ABCDh`
 `DX = 0`
 `ES:BX`

Points to buffer containing null-terminated command string

Output: `AX = 0`
 `AX = errcode`
 `AX = 102h`

No error; command performed satisfactorily
 Standard error code for RCL

Service not open; need `api_startcommands()`

Assembler

Example:

```
mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
mov     cx,6           ; Function call 6
push    ds
pop     es
ASSUME  ES:DATA
```

```

mov     bx,offset Reflection_command
int     21h
or      ax,ax
jnz     command_error

```

```
Reflection_command db "SEND ABC TO XYZ ASCII REC=80",0
```

C

```

int api_docommand(          /* Returns error code, 0, if no error */
    char *command_string    /* Points to null-terminated Reflection command */
);

```

Example:

```

#include "api.h"
main()
{
    /* Do a synch file transfer—print error code if failed */
    int  error;
    char *command_string = "SEND ABC TO XYZ ASCII REC=80";

    api_startcommands();
    if ( ( error = api_docommand( command_string ) ) !=0)
        printf( "Transfer failed, error code = %d\n",error );
    api_endcommands();
    return error;
}

```

BASIC

DECLARE FUNCTION api.docommand% CDECL (SEG rcommand\$)

Example—Transfer a file:

```

' $INCLUDE: 'api.inc'
'
i% = api.startcommands%
cm$ = "SEND ABC TO XYZ ASCII REC=80"
error% = api.docommand%( cm$ )
if error% > 0 then print "Transfer failed error code = ";error%
i% = api.endcommands%
end

```

Turbo Pascal

function api_docommand(var x:string) :integer;

Example—Send a file:

```
{ $V- }
uses    apiunit;
var     i      : integer;
        cm     : string[80];

begin
  cm := 'SEND ABC TO XYZ ASCII REC=80';
  if api_startcommands <> 0 then
    writeln ( 'Reflection busy' );
  if api_docommand( cm ) <> 0 then
    writeln ( 'error transferring file' );
  i := api_endcommands;
end.
```


api_endcommands

This function call stops a series of synchronous commands, and turns datacomm back on.

Input: CX = 10
AX = 2A52h
BP = ABCDh
DX = 0

Output: AX = 0 No error
 AX = errcode Should not return error if Reflection is present

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,10
int 21h
or ax,ax
jnz Serious_error
```

C

```
int api_endcommands( void );
```

For a C programming example, see `api_found` on page 186.

BASIC

```
DECLARE FUNCTION api.endcommands% CDECL ( )
```

For an example, see `api_startcommands` on page 250.

Turbo Pascal

```
function api_endcommands :integer;
```

For an example, see `api_startcommands` on page 250.

api_found

Return the value of the command language FOUND Boolean.

Input: CX = 9
AX = 2A52h
BP = ABCDh
DX = 0

Output:	AX =	Value of FOUND Boolean
	AX = 0	Not found
	AX <> 0	Found

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,9
int 21h           ; Call API
or ax,ax
jz String_Not_Found
```

C

```
int api_found( void );
```

Example:

```
#include "api.h"
main()
{
    /* Return the state of the FOUND Boolean */

    api_startcommands();

    /* Send username */

    api_docommand( "transmit 'john^m'" );
    api_docommand( "wait 0:0:8 for 'Password:'" );
```

```

    if ( !api_found() ) {
        printf( "Timed out waiting for password\n" );
    }
    api_endcommands();
    return 0;
}

```

BASIC

DECLARE FUNCTION api_found% CDECL ()

Example—Print a message if successful connect to remote modem:

```

' $INCLUDE: 'api.inc'
'
i% = api.startcommands
i% = api.docommand( "transmit 'ATDT 555-1212^m'" )
i% = api.docommand( "WAIT 0:0:45 for 'CONNECT'" )
if api_found then print "Connected to REMOTE MODEM"
i% = api.endcommands
end

```

Turbo Pascal

function api_found :integer;

Example—Print a message if successful connect to remote modem:

```

{$V-}
uses apiunit;
var cm : string[80];
    i : integer;
begin
    i := api_startcommands;
    cm := 'transmit "ATDT 555-1212^m"';
    i := api_docommand( cm );
    cm := 'WAIT 0:0:30 for "CONNECT"';
    i := api_docommand( cm );
    if api_found <> 0 then
        writeln ( 'Connected to REMOTE MODEM' )
    else
        writeln ( 'No connection' );
    i := api_endcommands;
end.

```


api_getkey

Return the current setting of a Reflection softkey to calling program. This function returns the value of Reflection 2 and Reflection 4 softkeys, Reflection 2 and Reflection 4 user-defined keys (UDKs), and Reflection 1 and Reflection 7 HP user keys.

Input:	CX = 22 AX = 2A52h BP = ABCDh DX = 0 ES:BX SI =	Point to buffer to return key Key number (1-relative) To return R2/R4 user-defined keys (UDKs) UDK # 6 = keynumber 17 ... UDK # 20 = keynumber 34
Output:	AX = 0 AX = 105h	No error; key record copied to buffer Error; bad key

HP User Key Label Structure (Reflection 1 and Reflection 7)

```

struct ukey {

    /* 0 = normal, 1 = local only, 2 = transmit only */
    char unsigned ukey_attr;

    /* Length of label */
    char unsigned ukey_lablen;

    /* Length of definition string */
    char unsigned ukey_deflen;
    int          ukey_reserved;

    /* Label string followed by definition */
    char          ukey_text[160];
};

```

DEC Softkey Structure (Reflection 2 and Reflection 4)

```

struct softkey {
    /* 0 = normal, 1 = local only */
    char unsigned action;
    char unsigned label_length;
    char label_text[8];
    char unsigned defn_length;
    char defn_text[80];
};

```

DEC UDK Structure (Reflection 2 and Reflection 4)

```

struct udk {
    char unsigned udk_length;
    char udk_text[255];
};

```

Assembler

Example:

```

mov     ax,02A52h                ; Get HP softkey # 3
mov     bp,0ABCDh
xor     dx,dx
push    ds
pop     es
mov     si,3                     ; User key # 3
ASSUME  ES:DATA
mov     bx,offset ukey_buffer    ; Where to put the ukey info
mov     cx,22                   ; Get user key
int     21h
or      ax,ax                   ; Error?
jnz     getkey_error             ; Yes

key_buffer label byte
ukey_attr db ?                  ; 0=normal, 1=local, 2=transmit only
ukey_lablen db ?                ; Length of label text
ukey_deflen db ?                ; Length of definition string
reserved dw ?
ukey_text db 16 dup (0)         ; Label text & definition start

```

C

```
int api_getfkey(
    struct ukey *key_buffer,
    int keynumber
);
```

Example:

```
#include "api.h"
main()
{
    /* Retrieve a function key and print the label */

    struct ukey key_buffer;
    int i;
    int keynumber=3;

    api_getfkey( &key_buffer, keynumber );
    i = key_buffer.ukey_lablen;

    /* Null-terminate after label */

    key_buffer.ukey_text[i]='\0';
    printf("key %d label '%s'\n", keynumber, key_buffer.ukey_text);
    return 0;
}
```

BASIC

DECLARE FUNCTION api.getfkey% CDECL
 (SEG keybuff AS ukeytype, BYVAL keyno%)

Example—Print the definition part of HP function key #3:

```
' $INCLUDE: 'api.inc'
'
COMMON x AS ukeytype
    i% = api.getfkey( x, 3 )
    y$ = x.ukeytext
    print MID$( y$, ASC( x.ukeylablen ) + 1 )
end
```


Turbo Pascal

function api_getfkey(k:integer;var fkey:ukeytype) :integer;

Terms: *k* Function key number

fkey User key record; see apiunit.pas for declaration

Example—Print the definition part of HP function key #3:

```
uses    apiunit;
var     x : ukeytype;                      (* Function key record *)
        i : integer;
        n : integer;                      (* Label length *)
        keydefn : string[80];
begin
    i := api_getfkey( 3, x );
    n := x.ukey_lablen;
    i := 0;
    keydefn[0] := chr(x.ukey_deflen);      (* Definition length *)
    while i < x.ukey_deflen do
        begin
            keydefn[i+1] := x.ukey_text[i + i ];
            i := i + 1;
        end;
    writeln ( 'definition of function key 3 = ',keydefn );
end.
```

api_getinfo

Gets information from Reflection. This call returns information of a relatively non-volatile nature, i.e., information that should not change from one second to the next.

Input:	CX = 3	
	AX = 2A52h	
	BP = ABCDh	
	DX = 0	
	ES:BX	Points to user buffer
	SI =	Number of words to be written to user buffer
		If SI exceeds structure size (25) remaining space is zeroed
Output:	AX = 0	No error; buffer initialized
	AX = errcode	Should not return error if Reflection present

Structure Definition

The integers labeled "HP" apply to Reflection 1 and Reflection 7; those labeled "DEC" apply to Reflection 2 and Reflection 4.

```

struct api_infostruc {
    int  apiinfo_apiversion;           /* Version number of API */
                                         /* High byte=minor version */
                                         /* Low byte=major version */

    int  apiinfo_function_key_mode;    /* # identifying current */
                                         /* set of function keys */
                                         /* displayed at bottom of */
                                         /* the Reflection screen */

    int  apiinfo_local_echo;
    int  apiinfo_remote_mode;
    int  apiinfo_caps_lock;
    int  apiinfo_display_functions;
    int  apiinfo_auto_linefeed;
    int  apiinfo_right_margin;
    int  apiinfo_phys_screen_width;
    int  apiinfo_memory_response;      /* HP */
    int  apiinfo_xmit_functions;       /* HP */
    int  apiinfo_spow_strap;           /* HP */

```

```

int  apiinfo_inheolwrp;           /* HP */
int  apiinfo_line_page_mode;     /* HP */
int  apiinfo_inhhndshk;          /* HP */
int  apiinfo_inhdc2;             /* HP */
int  apiinfo_block_mode;         /* HP */
int  apiinfo_format_mode;        /* HP */
int  apiinfo_memory_lock;        /* HP */
int  apiinfo_type_ahead;         /* HP */
int  apiinfo_normal_cursor_key_mode; /* DEC */
int  apiinfo_numeric_keypad_mode; /* DEC */
int  apiinfo_multipage_mode;     /* DEC */
int  apiinfo_user_features_locked; /* DEC */
int  apiinfo_udks_locked;        /* DEC */
};

```

Identifying Numbers

The numbers used to identify current function key mode, grouped by product, are listed below.

Reflection 2 and Reflection 4 Keys

As of version 4.2, Reflection 2 and Reflection 4 do not include device control, device modes, or “to” device function key sets. These sets have been replaced by logging keys. The obsolete sets had these names and identifying numbers: Device Control Keys–5, Device Modes Keys–6, and To Device Keys–7.

Function Key Set	Identifying Number
Main Menu	0
Softkeys	1
VT102 Keys	2
No Keys	3
Config Keys	4
Logging Keys	5
File Xfer Keys	8
Tab Keys	9
Terminal Keys	10
Graphics Keys	11

Reflection 1 and Reflection 7 Keys

The function key sets listed here include both the HP and VT emulations in Reflection 1 and Reflection 7.

Function Key Set	Identifying Number
Modes Keys	0
User Keys	1
System Keys	2
Config Keys	3
Margin/Tab Keys	4
Enhance Keys	5
Device Control Keys	6
Device Modes Keys	7
User Menu Keys	8
File Xfer Keys	9
To Device Keys	10
VT102 Keys	11*
VT Modes Keys	12*
VT Main Menu	13*
VT Tab Keys	14*
Define Fields Keys	15
Enhance Video Keys	16
User Keys 9–12	17

0 and 1 Settings

API can determine what the setting is for these Reflection configuration items:

Command	Set to 0	Set to 1
apiinfo_local_echo	OFF	ON
apiinfo_remote_mode	OFF	ON
apiinfo_caps_lock	OFF	ON
apiinfo_display_functions	OFF	ON
apiinfo_auto_linefeed	OFF	ON

* Settings when running VT emulation (R1V/R7V).

Command	Set to 0	Set to 1
apiinfo_right_margin	(0-relative)	
logical right hand margin of screen		
apiinfo_phys_screen_width	(1-relative)	
actual physical width of screen		
apiinfo_memory_response	0 = 4K 2 = 12K	1 = 8K 3 = 15K
apiinfo_xmit_functions	OFF	ON
apiinfo_spow_strap	OFF	ON
apiinfo_inheolwrp	OFF	ON
apiinfo_line_page_mode	LINE	PAGE
apiinfo_inhhndshk	OFF	ON
apiinfo_inhdc2	OFF	ON
apiinfo_block_mode	OFF	ON
apiinfo_format_mode	OFF	ON
apiinfo_memory_lock	OFF	ON
apiinfo_type_ahead	OFF	ON
apiinfo_normal_cursor_key_mode	APPL	NORMAL
apiinfo_numeric_keypad_mode	APPL	NORMAL
apiinfo_multipage_mode	OFF	ON
apiinfo_user_features_locked	UNLOCKED	LOCKED
apiinfo_udks_locked	UNLOCKED	LOCKED

Assembler

Example:

```

mov     ax, 02A52h
mov     bp, 0ABCDh
xor     dx, dx
push    ds
pop     es
ASSUME  ES:DATA
mov     bx, offset infostructure
mov     cx, 3
int     21h
or      ax, ax
jnz     info_error

```

C

```
int api_getinfo(
    struct api_infostruc *infobuffer,    /* Points to structure defined above */
    int count                            /* Size of infobuffer in words */
);
```

Example:

```
#include "api.h"
main()
{
    /* Find the physical screen width */

    struct api_infostruc infobuffer;
    struct api_infostruc *sptr;
    sptr = &infobuffer;

    api_getinfo( sptr , 25 );
    printf("Physical screen width = %d\n",
        sptr->apiinfo_phys_screen_width);
    return 0;
}
```

BASIC

DECLARE FUNCTION api.getinfo% CDECL
(SEG buffer AS RINFO, BYVAL rvarlen%)

Terms: *rvarlen%* Size of buffer

See api.inc for RINFO user-defined type definition.

Example—Get Reflection physical screen width:

```
' $INCLUDE: 'api.inc'
'
' See API.INC for description of user-defined type RINFO
COMMON X AS RINFO
'
i% = api.getinfo( X, 25% )
print "Reflection screen width is "; x.rphyscreenwidth
end
```


Turbo Pascal

function api_getinfo(i:integer;var x:info_array):integer;

Terms: *i* Size of info_array buffer
 info_array See apiunit.pas for info_array type declaration

Example—Get screen width:

```
uses    apiunit;
var     info : info_array;
        i : integer;

begin
  if api_getinfo( 25, info ) > 0 then exit;
  writeln( 'screen width is ', info[8] );
end.
```

api_getstatus

Return status of volatile information from Reflection. This information is time critical.

Input:	CX = 4	
	AX = 2A52h	
	BP = ABCDh	
	DX = 0	
	ES:BX	Point to structure
	SI	Specifies number of words (maximum 12) to be returned to user buffer. If SI is greater than structure size, excess buffer area is zeroed.
Output:	AX = 0	No error
	AX = errcode	Should not return error if Reflection is present

Structure Definition

```

struct api_statstruc {
    int  apistat_pagetop_row;
    int  apistat_cursor_row;
    int  apistat_cursor_col;
    int  apistat_left_border;
    int  apistat_cursor_physrow;
    int  apistat_cursor_physcol;
    int  apistat_kb_lock_sw;
    int  apistat_batch_flag;
    int  apistat_datacomm_error_flag;      /* HP */
    int  apistat_host_prompt_received;     /* HP */
    int  apistat_xfer_pending_sw;          /* HP */
};

```

apistat_pagetop_row	Row number (0-relative) of display memory for top of screen, i.e., if 100 lines of text have scrolled off the top of the screen (and are still in display memory), the top of the screen would be display row 101 and api_pagetop_row 100.
apistat_cursor_row	Logical cursor row (0-relative) from beginning of display memory.
apistat_cursor_col	Logical cursor column (0-relative) from left hand margin of screen.
apistat_left_border	Column position of left border of screen (true left margin may have been scrolled off of screen).
apistat_cursor_physrow	Actual physical row where cursor is located (0-relative).
apistat_cursor_physcol	Actual physical column where cursor is located (0-relative).
apistat_kb_lock_sw	Keyboard lock status: 0 = No lock <>0 = Keyboard locked.
apistat_batch_flag	0 = Reflection IDLE <>0 = BUSY.
apistat_datacomm_error_flag	1 = DATACOMM ERROR has occurred since last primary status request.
apistat_host_prompt_received	1 = Host prompt has been received. 0 = Waiting for host prompt.
apistat_xfer_pending_sw	<>0 = Transfer of data has been requested from host. Waiting to receive prompt before starting transfer. 0 = No transfer pending.

Assembler

Example:

```

mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
push    ds
pop     es
ASSUME  ES:DATA
mov     bx,offset status_struct
mov     cx,4
int     21h
or      ax,ax
jnz     info_error

```

status_struct dw 22 dup (?) ; 44 byte buffer to receive data

C

```

int api_getstatus(
    struct api_statstruc *statusbuf, /* statusbuf pointer to structure */
    int siz
);

```

Example:

```

#include "api.h"
main()
{
    /* Find which row the cursor is on */

    struct api_statstruc buffer;
    struct api_statstruc *sptr;
    sptr = &buffer;

    /* Returns 11 status info words */

    api_getstatus(sptr, 11);
    printf( "cursor row = %d", sptr->apistat_cursor_row );
    return 0;
}

```

BASIC

DECLARE FUNCTION api.getstatus% CDECL
(SEG buffer AS RSTAT, BYVAL rvarlen%)

Terms: *rvarlen%* Size of buffer

See api.inc for RSTAT user-defined type definition.

Example—Get physical cursor and row position:

```
' $INCLUDE: 'api.inc'
'
' See API.INC for description of user-defined type RSTAT
COMMON Y AS RSTAT
'
i% = api.getstatus( Y, 11% )
print "Cursor at ROW ";y.Rcursorphysrow ;" COL ";y.Rcursorphyscol
end
```

Turbo Pascal

function api_getstatus(i:integer;var x:stat_array):integer;

Terms: *i* Size of stat_array buffer
 stat_array See apiunit.pas for stat_array type declaration

Example:

```
uses    apiunit;
var     status : stat_array;

begin
  if api_getstatus( 11,status ) > 0 then exit;
  writeln ( 'cursor row is ',status[1], ' column ',status[2]);
end.
```

api_getvar

Return contents of variable to user buffer. Use this command only as part of a synchronous command sequence, otherwise it can return random values. If it interrupts Reflection's changing of one variable to another, a false result could occur.

Note: A Reflection variable may contain characters of any value including nulls. Variables cannot always be null-terminated since it is possible for them to contain nulls. ▲

Input:	CX = 7	
	AX = 2A52h	
	BP = ABCDh	
	DX = 0	
	ES:BX	Points to 80 byte buffer
	SI =	Variable number
Output:	AX = 0	No error
	SI = #	No error; length of variable
	AX = 10Ah	Error; bad variable number

Assembler

Example—Get variable V8 and null-terminate:

```

mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
mov     cx,7                      ; Function GetVar
mov     si,8                      ; Get variable V8
push    ds
pop     es
ASSUME  ES:DATA
mov     bx,offset Variable_Buffer
int     21h
mov     byte ptr [variable_length+SI],0 ; SI=length of V8 - put
                                           ; NULL at end of string
or      ax,ax                     ; Was there an error?
jnz     Bad_variable_number       ; Yes

```

```
Variable_Buffer db 80 dup (?)
```


C

```

int api_getvar(                                /* Returns length of variable */
                                /* Returns non-zero if error */
    char *var_buffer,                        /* Buffer for 80 byte variable */
    int varno,                               /* Variable number 0-799 */
    int *varlength                          /* Returned length of variable */
);

```

Example:

```

#include "api.h"
main()
{
    /* Find out if a file is present on the HP 3000 */

    int length;
    char varbuf[81];

    api_startcommands();
    printf( "Enter filename: " );
    gets( varbuf );
    length = strlen( varbuf );
    api_setvar( varbuf, 1, length );
    api_docommand( "transmit 'listf $1^m'" );
    api_docommand( "readhost 0:0:5 v2" );

    /* Get HP response */
    api_docommand( "readhost 0:0:5 v2" );
    api_docommand( "wait 0:0:8 for '^q'" );

    /* Get v2 */
    api_getvar( varbuf, 2, &length );

    /* Null-terminate v2 */
    varbuf[length] = '\0';
    if ( strstr( varbuf, "CIERR" ) )
        printf( "File not found on host\n" );
    api_endcommands();
    return 0;
}

```

BASIC

DECLARE FUNCTION api.getvar% CDECL
 (SEG var\$, BYVAL varnumber%, SEG rvarlen%)

Terms:	<i>var\$</i>	String first initialized to at least 80 characters in size for receiving variable data
	<i>varnumber%</i>	Variable number
	<i>rvarlen%</i>	Length of variable

Example—Find out if a file is present on a VAX:

```
' $INCLUDE: 'api.inc'
'
LINE Input: "Enter VAX filename"; f$
if f$< "!" then end
i% = api.startcommands
j$ = "transmit 'dir '" + f$ + CHR$( 13 ) + "' "
i% = api.docommand( j$ )
i% = api.docommand( "readhost 0:0:5 v2" )
i% = api.docommand( "readhost 0:0:5 v2" )
i% = api.docommand( "readhost 0:0:5 v2" )
i% = api.docommand( "wait 0:0:8 for '^m$'" )
varbuf$ = space$( 80 )
i% = api.getvar( varbuf$, 2, 1% )
varbuf$ = MID$( varbuf$, 1, 1% )
if instr( varbuf$, "%DIRECT-W-NOFILES" ) > 0 then
  print "File "; x$; "not found on VAX"
else
  print "File "; x$; "on VAX"
end if
i% = api.endcommands
end
```

Turbo Pascal

function api_getvar(i:integer;var x:string) :integer;

Terms: *i* Variable number
 x String variable to return data to

Example—Find out if a file is present on a VAX:

```
uses    apiunit;
{$V-}
var     fname : string[80];
        cm : string[80];
        i : integer;
        vlen : integer;
begin
    write ( 'Enter VAX filename' );
    readln( fname );
    if fname < '!' then exit;
    i := api_startcommands;
    cm := 'transmit "dir ' + fname + '^m"';
    i := api_docommand( cm );
    cm := 'readhost 0:0:5 v2';
    i := api_docommand( cm );
    i := api_docommand( cm );
    i := api_docommand( cm );
    cm := 'wait 0:0:8 for "^m$"';
    i := api_docommand( cm );
    i := api_getvar( 2, cm );
    if pos( '%DIRECT-W-NOFILES', cm ) <> 0 then
        writeln ( 'File ', fname, ' not found on VAX' )
    else
        writeln ( 'File ', fname, ' on VAX' );
end.
```


api_instchk

See if the API support code has been loaded via the /W switch. Determines Reflection product, version, and serial numbers.

The remaining 17 bytes of the buffer are reserved for future use.

Input:	CX = 0	
	AX = 2A52h	
	BP = ABCDh	
	DX = 0	
	ES:BX	Point to 32 byte buffer to receive serial number
Output:	AX = 0	No error; null-terminated Reflection 14-byte serial number copied to buffer. See the <i>Command Language Manual</i> for the serial number format.
	AX <> 0	Error

Assembler

Example:

```

mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
mov     cx,0
push    ds
pop     es
ASSUME  ES:DATA
mov     bx,offset serialno_buffer      ; ES:BX -> buffer
int     21h
or      ax,ax
jnz     not_installed                  ; Error if AX not zero

serialno_buffer db 32 dup (0)
```

C

```
int api_instchk(          /* Returns 0 if API support present */
    char *serialbuf       /* serialbuf points to 32 byte buffer */
);
```

Example:

```
#include "api.h"
main()
{
    char serialbuf[32];

    if ( api_instchk( serialbuf ) ) {
        printf("API support not present; restart with /W switch\n");
        return 1;
    }
    else
        printf( "Reflection serial number %s\n", serialbuf );
    return 0;
}
```

BASIC

DECLARE FUNCTION api.instchk% CDECL (SEG sermo\$)

Set sermo\$ to spaces prior to call:

```
sermo$ = space$( 32 )
```

Example—Get the Reflection serial number if API is present:

```
' $INCLUDE: 'api.inc'
'
sermo$ = space$( 32 )
if api.rcheck%( sermo$ ) <> 0 then
    print "API support not present, restart with /W switch"
end if
```

Turbo Pascal

```
function api_instchk( var s:string ) :integer;
```

Example:

```
uses    apiunit;
var     serno : string[32];

begin
    if api_instchk( serno ) > 0 then exit;
    writeln( 'Serial number is ', serno )
end.
```


api_keystatus

Each key in the keyboard queue takes two bytes of space. This function returns the amount of free space remaining in the keyboard queue.

Input: CX = 12
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = *<Amount of free space in units of keys>*

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,12
int 21h
or ax,ax

jz No_free_space_remaining
```

C

```
int api_keystatus( void );
```

Example:

```
#include "api.h"
main()
{
    if ( !api_keystatus() )
        printf( "API key buffer full\n" );
    return 0;
}
```

BASIC

DECLARE FUNCTION api.keystatus% CDECL ()

Example—See if any room left to queue keys:

```
' $INCLUDE: 'api.inc'
'
if api.keystatus = 0 then print "API key buffer full"
```

Turbo Pascal

function api_keystatus :integer;

Example:

```
uses    apiunit;
var     i : integer;
begin
    if api_keystatus <> 0 then
        writeln( 'API key buffer full' );
end.
```

api_ndcpopup

Pop Reflection into the foreground and buffer any data arriving at the datacomm port. The API client application will be frozen until the user presses the hot-key or a queued BACKGROUND command is processed.

Input: CX = 37
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0 Should not return an error

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,37
int 21h                               ; This call freezes API client program
```

C

```
int api_ndcpopup( void );
```

Example:

```
#include "api.h"
main()
{
    api_ndcpopup();
    printf( "Back from POP COMMAND\n" );
    return 0;
}
```


BASIC

DECLARE FUNCTION api.ndcpopup% CDECL ()

Example—Pop up Reflection:

```
' $INCLUDE: 'api.inc'
'
i% = api.ndcpopup
print "BACK from Reflection"
```

Turbo Pascal

function api_ndcpopup :integer;

Example—Pop up Reflection:

```
uses   apiunit;
var    i : integer;
begin
    i := api_ndcpopup;
end.
```

api_offerbuf

This function provides larger buffers for command or keyboard queues than the default API buffers. The default buffer size is 196 bytes for the command buffer and 31 keys for the keyboard buffer.

Make sure that the current queue is empty before offering a new buffer, otherwise any queued keys or commands are lost. Do not modify the offered buffer space until it has been canceled (see below). Note that the pointer passed to API is a far pointer (32-bit address).

Input:	CX = 24	
	AX = 2A52h	
	BP = ABCDh	
	DX = 0	
	ES:BX	Points to buffer
	SI =	Size of buffer
	DI =	Buffer type
		0 = keyboard buffer, 1 = command queue buffer
Output:	AX = 0	No error
	AX = 103h	Error; bad buffer type

Assembler

Example:

```

mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
push    ds
pop     es
ASSUME  ES:DATA
mov     bx,offset command_buffer
mov     di,1                      ; 1 <==> buffer for command queue
mov     si,CBUFFER_LEN           ; Length of buffer
mov     cx,24
int     21h

command_buffer db 2000 dup (0)
CBUFFER_LEN   EQU $ - command_buffer

```

C

```
int api_offerbuf(
    char far *command_queue,
    int bufsize,
    int type
);
```

Example:

```
#include "api.h"

#define BUFSIZE      2000
#define BUFFER_TYPE  1
char far command_queue[BUFSIZE];

main()
{
    /* Offer a 2000 byte command buffer */

    api_offerbuf( command_queue, BUFSIZE, BUFFER_TYPE );
    api_cancelbuf();
    return 0;
}
```

BASIC

DECLARE FUNCTION api.offerbuf% CDECL
 (SEG buffer AS bufftype, BYVAL rvarlen%, BYVAL typ%)

Terms:	<i>buffer</i>	Buffer to be used for queuing keys or commands; see api.inc for user-defined type definition
	<i>rvarlen%</i>	Size of the buffer

Example—Set up 512 byte command buffer within BASIC's data segment and queue some commands:

```
' $INCLUDE: 'api.inc'
'
' See api.inc for bufftype declaration
COMMON cmdbuf as bufftype
'
i% = api.offerbuf( cmdbuf, 512, 1% )
i% = api.qcmd( "dir *.*" )
i% = api.qcmd( "display '^g'" )
i% = api.qcmd( "wait 0:0:10" )
i% = api.wait
i% = api.cancelbuf
end
```

Turbo Pascal

function api_offerbuf(type,buflen:integer;var x:buffer):integer;

Terms:	<i>type</i>	0 or 1 for keys or commands
	<i>buflen</i>	Size of buffer
	<i>x</i>	Buffer; see apiunit.pas for declaration

Example—Offer a keyboard queue buffer from Pascal's data space:

```
{$V-}
uses  apiunit, crt;
const maxlength = 256;
var   keys : string[250];
      i : integer;
      keybuf : buffer;
begin
  i := api_offerbuf( 0, 256, keybuf );
  keys := "enter these keys and press return" return ' ;
  i := api_qkeys( keys );
  repeat
  until keypressed or ( api_block = 0 );

  (* Cancel buffer before terminating pgm *)
  i := api_cancelbuf;
end.
```

api_popup

Pop Reflection into the foreground. API client application will be frozen until user presses hot-key or a queued BACKGROUND command is processed.

Input: CX = 20
AX = 2A52h
BP = ABCDh
DX = 0

Output: AX = 0 Should not return an error

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,20
int 21h                      ; This call freezes API client program
```

C

```
int api_popup( void );
```

Example:

```
#include "api.h"
main()
{
    api_popup();
    printf( "Back from POP COMMAND\n" );
    return 0;
}
```

BASIC

DECLARE FUNCTION api.popup% CDECL ()

Example—Pop up Reflection:

```
' $INCLUDE: 'api.inc'
,
i% = api.popup
print "BACK from Reflection"
```

Turbo Pascal

function api_popup :integer;

Example—Pop up Reflection:

```
uses    apiunit;
var     i : integer;
begin
    i := api_popup;
end.
```


api_qcmd

Queue a new command in the command queue. This call returns immediately to the caller and the command is processed asynchronously via background multitasking.

Input:	CX = 16	
	AX = 2A52h	
	BP = ABCDh	
	DX = 0	
	ES:BX	Point to null-terminated command
Output:	AX = 0	No error
	AX = 100h	Error; a DISABLE RCL command is in effect
	AX = 101h	Error; synchronous commands being issued
	AX = 104h	Error; queue full
	AX = 106h	Error; bad length

Assembler

Example:

```

mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
mov     cx,16
push    ds
pop     es
ASSUME  ES:DATA
mov     bx,offset command_string
int     21h
or      ax,ax
jnz     Command_Queue_error

```

```

Command_string db "Send ABC to XYZ ASCII DELETE",0

```

C

```
int api_qcmd(                /* Returns error code */
    char *command_string
);
```

For a C programming example, see the `api_cmdstatus` example on page 180.

BASIC

DECLARE FUNCTION api.qcmd% CDECL (SEG rcommand\$)

Example—Queue a command and report if there was an error:

```
' $INCLUDE: 'api.inc'
,
cm$ = "SEND ABC TO DEF ASCII REC = 256"
if api.qcmd( cm$ ) then print "Error queuing command"
```

Turbo Pascal

```
function api_qcmd( var cmd:string ) :integer;
```

Terms: *cmd* String containing command

Example:

```
uses     apiunit;
        var cm : string[80];
        i : integer;
begin
    cm := 'SEND ABC TO DEF ASCII REC = 256';
    if api_qcmd( cm ) <> 0 then
        writeln( 'Error queuing command' );
end.
```

api_qkeys

This function call queues Reflection keystrokes and returns immediately to the caller. Since this is an asynchronous (queued) call, the caller will not know when the keys are sent. The keys are entered into Reflection's keyboard buffer exactly as though they had been typed, except no remapping takes place.

There are two kinds of keys on your keyboard, those that cause standard ASCII characters to be transmitted and those that perform control functions like resetting an internal modem, bringing up a configuration screen, or displaying the softkeys. The `api_qkeys` function differentiates among these sets of keys. Keys that cause ASCII values to be transmitted or entered are passed by enclosing the string of keys in quotes—either single or double. For example:

```
"read mail"
"don't exit"
'he said "wait for me!" '
```

C uses quotes to delimit strings, but the quotes are not part of the string. For this reason, you may have to use single quotes within C string constants such as:

```
char *keyentry = "'find employee miller'"
```

Control keys such as RETURN and VT-ENTER are keywords and must be passed to the `api` function unquoted. These keyword definitions are the same as those used in Reflection's keyboard mapping utility (KEYMAP or KEYCOMP). Here's an example of entering a string of keys followed by a carriage return:

```
api_qkeys( "MyPassword' RETURN" );
```

The password `MyPassword` is a quoted string, but the keyword `RETURN` is not quoted. The key names are documented in "Keyboard Layouts," starting on page 49.

Input: CX = 13
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output:	AX = 0	No error
	AX = 104h	Error; queue full
	AX = 105h	Error; bad key

Assembler

Example:

```
mov    ax,02A52h
mov    bp,0ABCDh
xor    dx,dx
push   ds
pop     es
mov    bx,offset key_buffer
mov    cx,13
int    21h
or     ax,ax
jnz    key_queue_error
```

```
key_buffer db "'Hello user.acct' return",0
```

C

```
int api_qkeys(          /* Returns non-zero if error */
    char *keybuffer
);
```

Example:

```
#include "api.h"
main()
{
    /* Queue some keys if there is room in the key queue */

    char *keybuffer = "'Hello user.acct' return";
    if ( strlen( keybuffer ) <= api_keystatus() )
        api_qkeys( keybuffer );
    else {
        printf( "No room in Reflection key queue.\n" );
        return 1;
    }
    while( !api_block() )
        if ( kbhit() )
            break;
    return 0;
}
```

BASIC

DECLARE FUNCTION api.qkeys% CDECL (SEG keys\$)

Terms: *key\$* String containing keystrokes to queue

Example—Queue keys to log on to an HP 3000:

```
' $INCLUDE: 'api.inc'
'
i% = api.qkeys( "'hello user.acct' return" )
```

Turbo Pascal

function api_qkeys(var keys:string) :integer;

Terms: *keys* String containing keystrokes

Example:

```
{ $V- }
uses    apiunit;
var    keys : string[80];
       i : integer;

begin
      keys := "'hello user.acct" return';
      i := api_qkeys( keys );
end.
```

api_rcheck

See if a copy of Reflection is present in the background.

Input: AX = 2A57h
 CX = not used
 BP = ABCDh
 DX = 0

Output: AX = "RQ" Implies that Reflection is present
 AX = other Implies that Reflection is not present

This function departs from the standard call format: AX = 2A57h is used instead of 2A52h, and CX is not used. The older calling convention (AX = DE57h) still works, however.

This function may be used to detect the presence of any version of Reflection including those prior to version 3.4. It does not imply that the API code is present, only that a copy of Reflection is present in background. This will enable you to tell a user that, while Reflection may be installed, either the /W switch was not used, or it is a pre-3.40 version.

Assembler

Example:

```
mov  ah,ax
int  21h
cmp  ax, "RQ"                ; Does AX have correct signature?
jz   Reflection_present      ; Yes, Reflection is present
mov  ah,9                    ; No, print error message and exit
mov  dx,offset error_message
int  21h
mov  ah,4Ch
int  21h

error_message db "Reflection not installed",0Dh,0Ah,"$"
```


C

int api_rcheck(void); /* Returns 0 if Reflection is present */

Example:

```
#include "api.h"
main()
{
    /* See if Reflection is in background */

    if ( api_rcheck() ) {
        printf( "Reflection not installed\n" );
        return 1;
    }
    else {
        printf( "Reflection in background\n" );
        return 0;
    }
}
```

BASIC

DECLARE FUNCTION api.rcheck% CDECL ()

Example—See if Reflection is installed:

```
' $INCLUDE: 'api.inc'
'
if api.rcheck <> 0 then
    print "Reflection not installed"
else
    print "Reflection in background"
end if
```

Turbo Pascal

function api_rcheck :integer;

Example—See if Reflection is installed:

```
uses   apiunit;
var
begin
    if api_rcheck <> 0 then
        writeln( 'Reflection not installed' )
    else
        writeln( 'Reflection in background' );
end.
```

api_rdchar

Directly read asynchronous datacomm from Reflection's receive buffer. Reflection will not read the characters. The function `api_startcommands()` must be invoked first to stop Reflection from reading its incoming datacomm.

It is the API client program's responsibility to keep up with the incoming data. Make sure that the appropriate flow control is set. Setting RECEIVE-PACING to XON/XOFF is recommended in most cases. Character translation takes place from the host character set to the PC character set unless SET DISABLE-TRANSLATION YES is in force.

Input:	CX = 33 AX = 2A52h BP = ABCDh DX = 0 ES:BX	Points to 2 byte buffer for storing character (null is second byte)
Output:	AX = 0 AX <> 0	No error; character read Error; no characters available

Assembler

Example:

```

mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
push    ds
pop     es
ASSUME  ES:DATA
mov     bx,offset receive_buf
mov     cx,33
int     21h
or      ax,ax
jz      received_a_character

```

```
receive_buf db ?,0
```


C

```
int api_rdchar(          /* Read character and store in *string */
    char *string        /* Returns NZ if character not available */
);
```

Example:

```
#include "api.h"
main()
{
    /* Transmit some characters and read their echo */

    char one_byte[2];
    char temp[33];
    int k;

    /* Tell Reflection not to read incoming data */

    if ( api_startcommands() ) {
        printf( "can't start\n" );
        exit();
    }
    k=0;
    api_docommand("transmit 'Read echo from these characters^m'");
    while( k < 33 ) {
        if ( kbhit() )
            break;
        if ( !api_rdchar( one_byte ) )
            temp[k++]=*one_byte;
    }
    temp[k]='\0';
    printf( "%s\n",temp );
    api_endcommands();
    return 0;
}
```

BASIC

DECLARE FUNCTION api_rdchar% CDECL (SEG char\$)

Terms: *char\$* Must be initialized to a length of 1 prior to call—if found, *char\$* will contain character that has been read and *api_rdchar%* = 0

Example—Send a character and then read the echo coming back:

```
' $INCLUDE: 'api.inc'
'
ch$ = space$( 1 )
i% = api.startcommands
'
' Send a character
' Depending on host, will probably be echoed
i% = api.writeasync( asc("a") )
'
' rdloop waiting for character to come back
' or user to press key
do
loop while api.rdchar( ch$ ) <> 0 and inkey$=""
if ch$ = "a" then
    print "Successfully transmitted and received character"
i% = api.endcommands
end
```

Turbo Pascal

function api_rdchar(var x:integer) :integer;

Terms: *x* Integer that will contain character after successful read (*api_rdchar* returns 0)

Example—Write a string and read back the echo and print it out (requires connection to full duplex host with local echo off):

```
{ $V- }
uses    apiunit;
var     j : integer;
        k : integer;
        i : integer;
        srctext : string[255];
        destext : string[255];
        c : integer;

begin
    srctext := 'send this string';
    destext := '';
    i := api_startcommands;
    k := 0;
    while k < length( srctext ) do
        begin
            if api_xmitstatus > 0 then
                begin
                    j := api_writeasync( ord( srctext[k+1] ) );
                    k := succ( k );
                end;
            end;
        k := 0;
        while k < length( srctext ) do
            if api_Rdchar ( c ) = 0 then
                begin
                    destext[k+1] := chr( c );
                    k := succ( k );
                end;
            destext[0] := chr( k );
            j := api_endcommands;
            writeln( 'received data = ', destext );
        end.
```


api_releasedc

The purpose of this command is to stop Reflection from taking datacomm hardware away from the foreground process. Normally if Reflection is running a command file or processing a command, it will take the datacomm hardware (COM1, COM2, etc.) away from a foreground process. This call allows a foreground task to initialize the datacomm hardware and do direct datacomm I/O without interference from Reflection in background. Reflection will not be able to do datacomm.

This call will not restore datacomm to the foreground program if it has already been taken away. The foreground program will have to re-initialize the datacomm hardware. The `api_releasedc` function should only be required when the hardware serial ports are being used by both Reflection and a foreground program.

Input: CX = 18
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0 Should not return an error

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,18
int 21h
```

C

```
int api_releasedc( void );
```

Example:

```
#include "api.h"
main()
{
    api_releasedc();
    return 0;
}
```

BASIC

DECLARE FUNCTION api.releasedc% CDECL ()

Example—Release datacomm:

```
' $INCLUDE: 'api.inc'  
,  
i% = api.releasedc
```

Turbo Pascal

function api_releasedc :integer;

Example:

```
i := api_releasedc
```

api_reset

Sends Reflection a hard reset. See the *Reflection Technical Reference* for a description of the hard reset process. In addition, `api_reset` performs the following three functions:

- ▲ `api_clrcmd()`
- ▲ `api_clrkeybd()`
- ▲ `api_endcommands()`

To prevent the host from performing a hard-reset, issue the command `SET EXITS-DISABLED YES`.

Input: CX = 21
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0 Should not return an error

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,21
int 21h
```


C

```
int api_reset( void );
```

Example:

```
#include "api.h"
main()
{
    int status;
    printf( "Resetting Reflection.\n" );
    status = api_reset();
    if ( status )
        printf( "An error occurred. Error code is: %d\n", status );
    return status;
}
```

BASIC

DECLARE FUNCTION api.reset% CDECL ()

Example—Hard reset Reflection:

```
' $INCLUDE: 'api.inc'
'
i% = api.reset
```

Turbo Pascal

```
function api_reset :integer;
```

Example—Hard reset Reflection:

```
uses    apiunit;
var     i : integer;

begin
    i := api_reset;
end.
```

api_screenread

Read text from the Reflection screen in background.

Screen reads do not extend to the function key area. Returns an error if the screen is in graphics mode.

Input:	CX = 11	
	AX = 2A52h	
	BP = ABCDh	
	DX = 0	
	BL = ROW	0-relative
	BH = COL	0-relative
	ES:DI	Points to buffer to contain screen text
	SI =	Number of bytes to read
Output:	AX = 0	No error
	AX = 103h	Error; starting read location is off screen
	AX = 106h	Error; screen in graphics mode

Assembler

Example:

```

mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
mov     bl,[start_row]
mov     bh,[start_column]
mov     si,[bytes_to_read]
push    ds
pop     es
ASSUME  ES:DATA
mov     di,offset screen_buffer
mov     cx,11
int     21h
or      ax,ax
jnz     screen_read_error

```

C

```
int api_screenclear(          /* Returns non-zero if error */
    char *buffer,
    int row,
    int col,
    int length
);
```

Example:

```
#include "api.h"
main()
{
    /* Read a string off of the screen */

    char screen_buffer[30];
    api_startcommands();

    /* Home cursor, clear screen, position to row 4, column 6 */
    /* NOTE-these are HP escape sequences! */

    api_docommand( "display '^[H^[J^[&a4r6C" );

    /* Display sample text */
    api_docommand( "display 'line at row 4 column 6'" );

    /* Read 22 bytes off of screen at row 4 column 6 */
    api_screenclear( screen_buffer, 4, 6, 22 );

    api_endcommands();

    screen_buffer[22]='\0';
    if ( strcmp( screen_buffer,"line at row 4 column 6" ) ) {
        printf( "returned string doesn't compare\n" );
        return 1;
    }
    else
        printf( "Found [%s]\n",screen_buffer );
    return 0;
}
```


BASIC

DECLARE FUNCTION api.screenread% CDECL
(SEG scrbuffer\$, BYVAL row%, BYVAL col%, BYVAL rvarlen%)

Terms:	<i>scrbuffer\$</i>	Buffer to receive screen data— <i>scrbuffer\$</i> must first be initialized to size sufficient to contain data
	<i>row%</i>	Row to start read
	<i>col%</i>	Column to start read
	<i>rvarlen%</i>	Number of bytes to read

Example—Print Reflection screen (characters only, no color):

```
' $INCLUDE: 'api.inc'
'
' Print the Reflection screen.
'

scbuf$ = space$( 80*24 )
i% = api.screenread( scbuf$, 0%, 0%, 80%*24% )
cls
print scbuf$
end
```

Turbo Pascal

function api_screenread(length,column,row:integer;var x:buffer):integer;

Terms:	<i>length</i>	Number of bytes to read
	<i>column</i>	Column to start read
	<i>row</i>	Row to start read
	<i>x</i>	Buffer to receive data. See apiunit.pas for declaration

Example—Read and print 80 columns from Reflection screen:

```
{ $V- }
uses    apiunit;
var     scrbuf : buffer ;
        j, row, col, k, i : integer;
        scrtext : string[255];
begin
    write( 'enter row and column to start screen read:' );
    readln( row, col );
    j := 80;
    if api_screenread( j, col, row, scrbuf ) <> 0 then exit;
    k := 1;
    scrtext[0] := chr( j );
    while k <= j do
        begin
            scrtext[k] := scrbuf[k-1];
            k := k+1;
        end;
    writeln( scrtext );
end.
```

api_searchscreen

Search the Reflection screen for a string. If found, return the row and column location where the string starts.

Input: CX = 32

AX = 2A52h

BP = ABCDh

DX = 0

BH = Column (0-relative) where to start search

BL = Row (0-relative)

ES:DI Buffer containing null-terminated string to seek

Output: AX = 0 No error; found string

BH = No error; column (0-relative)

BL = No error; row (0-relative)

AX <> 0 Error; string not found, bad coordinates, or zero-length string

Assembler

Example:

```

mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
push    ds
pop     es
ASSUME  ES:DATA
mov     bl,[row]
mov     bh,[column]
mov     di,offset search_string    ; String to search for
mov     cx,32
int     21h
or      ax,ax
jnz     string_not_found
mov     [row],bl                  ; Store location where found
mov     [column],bh

column      db 20
row         db 2
search_string db "Enter Password:"

```


C

```
int api_searchscreen(      /* Returns non-zero if not found */
    char *search_string,
    int *row,
    int *column
);
```

Example:

```
#include "api.h"
main()
{
    /* Search screen for field. If found, print contents */

    /* Search screen for a prompt */

    int row, col;
    char *string = "Employee Name: ";
    char employee[30];

    col = 0;
    row = 0;
    if ( api_searchscreen( string, &row, &col ) ) {
        printf( "Can't find employee field\n" );
        return 1;
    }
    else {

        /* Skip 'Employee Name:' */

        col = col + 15;
        api_screenread( employee, row, col, 30 );
        employee[30] = '\0';
        printf( "Employee ( %s )\n", employee );
    }
    return 0;
}
```

BASIC

DECLARE FUNCTION api.searchscreen% CDECL
(SEG srchstrng\$, SEG row%, SEG col%)

Terms:	<i>srchstrng\$</i>	String being searched for
	<i>row%</i>	Row where search is to begin
	<i>col%</i>	Column where search is to begin—if found, row% and col% are set to the location of string

Example—Search screen for field. If found, print contents:

```
' $INCLUDE: 'api.inc'
,
srchtext$ = "Employee Name:"
,
' Allow 40 spaces for name variable
namevar$ = space$( 40 )
column% = 0
row% = 0
if api.searchscreen( srchtext$, row%, column% ) <> 0 then
    print "Can't find employee field"
else
    print "found", row%, column%
    column% = column% + 15
    i = api.screenread( namevar$, row%, column%, 40 )
    print "Employee name : "; namevar$
end if
end
```

Turbo Pascal

function api_searchscreen(var col,row:integer;var x:string) :integer;

Terms:	<i>col</i>	Starting column
	<i>row</i>	Starting row
	<i>x</i>	String to search for—if string is found, column and row contain string position on screen

Example—Search screen for field. If found, print contents:

```
{ $V- }
uses   apiunit;
var    searchtxt : string[30];
        row : integer;
        col : integer;
        i : integer;
        namvar : string[40];
        scrtext : buffer;

begin
    row := 0;
    col := 0;
    searchtxt := 'Employee Name: ';
    if api_searchscreen( col, row, searchtxt ) <> 0 then
        writeln( 'employee field not found' )
    else
        begin
            col := col + 15;
            i := api_screenread( 40, col, row, scrtext );
            i := 0;
            repeat
                namvar[i+1] := scrtext[i];
                i := i + 1;
            until i = 40;
            namvar[0] := chr( 40 );
            writeln( 'Employee name: ' , namvar );
        end;
    end.
```


api_setfkey

Directly set an R1/R7 HP user key or an R2/R4 softkey. DEC UDKs can only be set via the DISPLAY command with the appropriate escape sequence.

Input: CX = 23
 AX = 2A52h
 BP = ABCDh
 DX = 0
 ES:BX Point to ukey/softkey structure
 SI = key number

Output: AX = 0 No error
 AX = 105h Error; bad key

Assembler

Example:

```
mov     ax,02A52h
mov     bp,0ABCDh
xor     dx,dx
push    ds
pop     es
ASSUME  ES:DATA
mov     bx,offset key_structure
mov     si,3                ; Set user key 3
mov     cx,23
int     21h
or      ax,ax
jnz     key_error

key_structure label byte
ukey_attr    db 0            ; Normal (as if typed at keyboard)
ukey_lablen  db 16          ; Length of label text (2 rows)
ukey_deflen  db 23          ; Length of definition string
reserved     dw ?
ukey_text    db ' LOGON ',' HP ','HELLO George,mgr.sales',0dh
```

C

```
int api_setfkey(  
    struct ukey *key_buffer,  
    int keynumber  
);
```

Example:

```
#include "api.h"  
main()  
{  
    /* Change function key 3 to LOCAL */  
  
    int keynumber = 3;  
    struct ukey key_buffer;  
    struct ukey *sptr;  
  
    sptr = &key_buffer;  
    api_getfkey( sptr, keynumber );  
    sptr->ukey_attr = 1;          /* 1 = local */  
    api_setfkey( sptr, keynumber );  
    printf( "Userkey 3 set to LOCAL [L] \n" );  
    return 0;  
}
```

BASIC

See api_getfkey on page 188.

Turbo Pascal

See api_getfkey on page 188.

api_setgetintstate

The `api_setgetintstate` function allows you to control whether an `api_docommand` is interruptible by the user. If the `api_setgetintstate` function is enabled, a user can interrupt a command that is currently being executed by pressing **Ctrl-Break**. This keystroke has the same effect as pressing **Ctrl-Y** during command file operation. To provide compatibility with previous versions of Reflection that did not support this function call, API defaults to not being interruptible. This means that it is possible for an application to hang if, for example, you are waiting for a string to arrive from the host with no timeout and the string never arrives. If you enable interruptibility, and a user presses **Ctrl-Break** during execution of an `api_docommand` function, you will receive an error code of 17 ("Terminated by user").

Input:	CX = 36	
	AX = 2A52h	
	BP = ABCDh	
	DX = 0	
	SI =	Desired state
		1 = enable breakout
		0 = disable breakout

Output:	AX =	Previous breakout state
		1 = breakout enabled
		0 = breakout disabled

Assembler

Example:

```

mov    si, 1                ; Enable breakout
mov    ax, 02A52h
mov    bp, 0ABCDh
xor    dx, dx
mov    cx, 36
int    21h
mov    [prev_state], ax     ; Save previous breakout state

```


C

```
int api_setgetintstate(  
    int state  
);
```

Example:

```
main()  
{  
    int old_state;  
    int err;  
  
    api_startcommands();  
  
    /* Save original interrupt */  
    /* state and enable ctrl-break */  
  
    old_state = api_setgetintstate( 1 );  
  
    /* Oops, too long to wait! */  
    /* Press ctrl-break here */  
  
    err = api_docommand( "wait 1:0:0" );  
  
    /* Restore original interrupt state */  
  
    api_setgetintstate( old_state );  
    api_endcommands();  
  
    /* If ctrl-break was pressed, a value of 17 is returned */  
  
    return err;  
}
```

BASIC

DECLARE FUNCTION api.setgetintstate% CDECL (BYVAL state%)

Terms: *state%* Desired interrupt state

Example:

```
' $INCLUDE: 'api.inc'
'
i% = api.setgetintstate(1%)
print "ctrl-break enabled"
```

Turbo Pascal

function api_setgetintstate(i:integer) :integer;

Example—Enable Ctrl-Break interrupt:

```
uses   apiunit;
var    i : integer;
begin
     i := api_setgetintstate(1);
     writeln ( 'ctrl-break enabled' );
end.
```

api_setvar

Load ASCII string into Reflection variable. Reflection variables are limited in length to 80 characters. The default number of variables is 10, but can be increased via a SET command to 800.

This command should only be executed as part of a synchronous command sequence, otherwise it can return suspect values, i.e., it may interrupt Reflection changing one variable to another so a false result could occur.

Variables cannot always be null-terminated in this way, since it is possible for them to contain nulls themselves.

Input:	CX = 8	
	AX = 2A52h	
	BP = ABCDh	
	DX = 0	
	ES:BX	Points to 80 byte buffer
	SI =	Variable number
	DI =	Length of variable
Output:	AX = 0	No error
	AX = 10Ah	Error; bad variable number
	AX = 106h	Error; bad length

Assembler

Example—Set variable V2 to value below:

```

mov     ax, 02A52h
mov     bp, 0ABCDh
xor     dx, dx
mov     cx, 8
mov     si, 2
mov     di, VARLENGTH
push    ds
pop     es
ASSUME  ES:DATA
mov     bx, offset Variable_Buffer
int     21h

mov     byte ptr [variable_length+SI], 0
or      ax, ax                      ; Was there an error?
jnz     Bad_variable_number        ; Yes

Variable_Buffer db "Enter this string in V2"
VARLENGTH      EQU $ - Variable_Buffer

```

C

```

int api_setvar(                      /* Returns non-zero if error */
    char *var_buffer,
    int varno,                       /* Variable number 0-799 */
    int varlength
);

```

For a C programming example, see `api_getvar` example on page 202.

BASIC

DECLARE FUNCTION api.setvar% CDECL (SEG var\$, BYVAL varnumber%)

Terms:	<i>var\$</i>	String containing data to load into variable
	<i>varnumber\$</i>	Variable number

Example—Set a command language variable:

```
' $INCLUDE: 'api.inc'
'
' a$ contains a null
a$ = "abc" + chr$(0) + "def"
'
' Set V3 to a variable that contains a null
i%=api.setvar(a$,3)
```

Turbo Pascal

function api_setvar(i:integer;var x:string):integer;

Terms:	<i>i</i>	Variable number
	<i>x</i>	String data to load into Reflection variable

Example—Set a command language variable:

```
{ $V- }
uses apiunit;
var cmdvar : string[80];
    retvar : string[80];
    i : integer;
begin
    cmdvar := 'Load this string in V3';
    i := api_setvar( 3, cmdvar );
    i := api_getvar( 3, retvar );
    writeln ( retvar );
end.
```

api_startcommands

Prepares Reflection for a series of synchronous commands. This call returns an error if any of the following conditions exist:

- ▲ Reflection is already performing a command
- ▲ Reflection is performing a file transfer
- ▲ A `DISABLE RCL` command is in effect
- ▲ A menu, dialog box, or the command line is open

If successful, this call turns off incoming datacomm (characters remain in receive buffer) unless an `api_docommand` is actually being processed.

After the `api_startcommands` function is issued, Reflection no longer reads incoming datacomm out of the receive buffer unless a Reflection command is actually being executed. In most cases, startcommands should not be issued until you are ready to start processing commands. Depending on the type of receive pacing, you may run the risk of overflowing the receive buffer. Note that datacomm is turned back on while Reflection is in foreground. Upon return to background, datacomm is turned back off unless a command is active.

Input: CX = 5
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0
 AX = 1000h

No error

Error; Reflection is busy, a `DISABLE RCL` command is in effect, or a menu, dialog box, or the command line is open

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,5
int 21h
or ax,ax

jnz Reflection_Busy
```

C

```
int api_startcommands( void );
```

Example:

```
#include "api.h"
main()
{
    if ( api_startcommands() ) {
        printf( "Reflection busy\n" );
        return 1;
    }
    else
        printf( "Reflection ready for commands\n" );
    return 0;
}
```

BASIC

```
DECLARE FUNCTION api.startcommands% CDECL ( )
```

Example—See if we can start docommand sequence:

```
' $INCLUDE: 'api.inc'
'
if api.startcommands% > 0 then print "Reflection busy"
end
```

Turbo Pascal

function api_startcommands :integer;

Example:

```
uses    apiunit;
var     i : integer;
begin
    if api_startcommands <> 0 then
        writeln ( 'Reflection busy' );
end.
```

api_wait

Wait for Reflection to empty the command and/or keyboard queues. Does not return until queue is empty. When a series of commands are queued, but the API client program cannot continue until the commands have been completed, this call could be used. Handle with care: a keyboard lock can make the system hang.

Input: CX = 26
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = 0 Should not return an error

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,26
int 21h
```

C

```
int api_wait( void );
```

Example:

```
#include "api.h"
main()
{
    /* Queue commands and WAIT until complete */

    api_qcmd( "wait 0:0:5" );
    api_qcmd( "display '^g'" );
    api_wait();
    return 0;
}
```


BASIC

DECLARE FUNCTION api.wait% CDECL ()

Example—Queue commands and WAIT until complete:

```
' $INCLUDE: 'api.inc'
'
i% = api.qcmd( "WAIT 0:0:5" )
i% = api.qcmd( "display '^g'" )
i% = api.wait
```

Turbo Pascal

function api_wait :integer;

Example:

```
uses    apiunit;
var     i : integer;
var     cm : string[80];
begin
    cm := 'SEND ABC TO XYZ';
    i := api_qcmd( cm );

    (* Wait until command completed *)
    (* NOTE: may cause program to hang if command can't complete *)
    i := api_wait
end.
```

api_writeasync

Write a single character to the Reflection transmit buffer for transmission to the host.

Input: CX = 34
AX = 2A52h
BP = ABCDh
DX = 0
SI = character

Output: AX = 0 No error
 AX <> 0 Error; buffer full

Assembler

Example:

```
mov ax,02A52h
mov bp,0ABCDh
xor dx,dx
mov cx,34
mov si,'a'
int 21h
or ax,ax
jnz buffer_full
```

C

```
int api_writeasync(
    int unsigned c
);
```

/* Write character c. */
/* Return NZ if xmit buffer full */

Example:

```
#include "api.h"
main()
{
    /* Transmit the contents of the string */

    /* Write a string */

    int c;
    char *xmit_string = "Send this string";
    while ( ( c =*xmit_string++ ) != '\0' ) {
        while( !api_xmitstatus() )
            if ( kbhit() )
                break;
        api_writeasync( c );
    }
    return 0;
}
```

BASIC

DECLARE FUNCTION api.writeasync% CDECL (BYVAL c%)

Terms: c% The ordinal value of the character within the ASCII character set. This can be derived via the ASC function
 c% = asc(c\$).

For an example, see api_rdchar on page 226.

Turbo Pascal

function api_writeasync(c:integer):integer;

Terms: c Integer containing character to be transmitted

For an example, see api_rdchar on page 226.

api_xmitstatus

Return the amount of free space in the transmit buffer. Should be used before `api_writeasync` to prevent accidental overrun of the transmit buffer and loss of data.

Input: CX = 35
 AX = 2A52h
 BP = ABCDh
 DX = 0

Output: AX = *<Amount of free space>*

Assembler

Example:

```
mov  ax,02A52h
mov  bp,0ABCDh
xor   dx,dx
mov  cx,35
int  21h
or   ax,ax           ; Is there any free space?
jz   buffer_full     ; No
```

C

```
int api_xmitstatus( void ); /* Returns number of bytes of free space */
                               /* if xmit buffer is full */
```

For a C programming example, see `api_writeasync` on page 255.

BASIC

DECLARE FUNCTION api.xmitstatus% CDECL ()

Example—Send an ASCII file out the datacomm port without flow control:

```
' $INCLUDE: 'api.inc'
'
open "filename" for input as #1
readloop:
    if eof(1) then goto endit
    line input #1, x$
    for charpos = 1 to len( x$ )
    '
        Delay until transmitter ready
        do
            loop while api.xmitstatus = 0
            i%=api.writeasync( asc( mid$( x$,charpos, 1) ) )
        next charpos
    '
    Delay until transmitter ready
    do
        loop while api.xmitstatus = 0
    '
    Delay until transmitter ready
    i%=api.writeasync( 13% )
    '
    Delay after carriage return
    for j% = 1 to 100 : next j%
    goto readloop
endit:
    close ( 1 )
end
```

Turbo Pascal

function api_xmitstatus :integer;

For an example, see api_rdchar on page 226.

API Error Codes

Hex	Decimal	Mnemonic	Description
0	0	NoError	No error encountered
100h	256	NotAvailable	Attempt to do <code>api_startcommands</code> when Reflection was busy, <code>DISABLE RCL</code> command was in effect, or menu, dialog box, or command line was open. Also attempt to do <code>api_qcmd</code> when <code>DISABLE RCL</code> command was in effect.
101h	257	OpenForSynch	Attempt to do <code>api_qcmd</code> when opened for docommands; first issue an <code>api_endcommands</code> function.
102h	258	ServiceNotOpen	Attempt to do an <code>api_docommand</code> before first issuing an <code>api_startcommands</code> . Also an attempt to do an <code>api_rdchar</code> without doing an <code>api_startcommands</code> to turn off <code>datacomm</code> .
103h	259	IllegalFncCall	Call parameters are incorrect, i.e., a screen read is requested that goes off of the screen.
104h	260	QueueFull	Keyboard or command queue is full.
105h	261	BadKey	An attempt was made to queue a key name that was not recognized.
106h	262	BadLength	Attempt made to set a variable with length greater than 80 or a command with a length greater than 159.
109h	265	NotFound	<code>api_searchscreen</code> failed; character not found on <code>api_rdchar</code> .

Hex	Decimal	Mnemonic	Description
10Ah	266	BadVarNo	The variable number used exceeded the current highest legal variable number.
0FFFFh	-1	Uninstalled	Reflection was uninstalled. Also AH=DEH for assembler.
<other codes>			api_docommand returns standard Reflection error codes, which are below 256. See the <i>Command Language Manual</i> for a list of these error codes.

How to Write an API Application

It is assumed that you are already familiar with Reflection and have written Reflection command language programs. If you haven't, that is probably the best and easiest place to start. Automating user tasks that involve a host computer is complex: your command file has to be completely synchronized with the application on the host. Even simple jobs, such as a command file to log a user on to a host computer, are not necessarily trivial.

API adds another level of complexity because the Reflection screen is no longer visible to indicate timing problems or error messages. When things don't work, it can be difficult to figure out what is wrong. An API application has to communicate with Reflection, which in turn communicates with the host. Creating API applications for Reflection requires knowledge of all of the following:

- ▲ Host communication issues
- ▲ The host application
- ▲ Reflection (especially Reflection command language)
- ▲ The programming language in which the API is written

Queued vs. Synchronous Commands

Some API commands are queued (asynchronous) and some are synchronous. Of the API functions that are provided, the synchronous command support offered by the `api_docommand` function is the easiest to use, and usually the most appropriate for interactive applications.

Synchronous Commands

Synchronous functions require that the API application wait for the function to be completed before proceeding. This is how most programs operate. When a call is made, it must complete before another call is issued. Synchronous commands give the API application direct control over operations and when they occur. When control returns to the API application, the command has either completed, or a return code indicates what went wrong.

Asynchronous Commands

Asynchronous commands capitalize on the multitasking capability of Reflection. A queued function requests Reflection to do something as time permits. The request is processed when Reflection gets to it in the queue, and there's no guarantee exactly when it will be done. The advantage is that the function call returns to the API application immediately (as soon as the request is queued), so the API application can continue while Reflection works on the queued request.

For example, if you transfer a file to the host programmatically using a synchronous call, the following happens:

- ▲ The function is called
- ▲ The file is transferred
- ▲ The function returns with an error code when the transfer is complete

However, if the program must simultaneously interact with a user, you may want to simply issue the transfer request, and then return the API application's attention to the user. File transfers are unlike most system calls (write to file, read character from keyboard, and so on) in that they can take a long time. Because the user gets no visual feedback, it may appear that the PC is hung and needs to be rebooted.

By using queued commands, you can get around this problem and pop up the file transfer screen during the transfer so the user will at least know that something is happening. The code fragment on the next page demonstrates this. It queues three commands: a command to perform the file transfer, a command to get the error code into a variable, and a command to switch back to background. It then issues the `api_popup` command so the user can view the file transfer. As soon as the file transfer completes, the error code is captured and Reflection returns to the background.

While these commands are queued, the effect is synchronous since the file transfer is performed in the foreground. It is important to capture the error code since it is possible for the user to stop the transfer via the STOP TRANSFER key. If this happens, capturing the error code will help you find out why the transfer failed.

Here is an example of using a queued command:

```
char  error_text[81];
int   length;
api_qcmd( "SEND testall.c TO data1;p ASCII DELETE" );
api_qcmd( "LET V9=ERROR-CODE" );
api_qcmd( "BACKGROUND" );
api_popup();
api_getvar( error_text, 9 , &length );
error_text[length]='\0';
printf( "File transfer completed\n" );
printf( "Error code %s",error_text );
exit();
```

Queued File Transfer

When an API client queues a file transfer, the file transfer usually will not start immediately. Consequently, if you immediately test `apistat_batch_flag` to see if the transfer is complete (`== 0`), you will probably be testing it before Reflection has had a chance to turn it on. The best way to determine Reflection's status, and see whether the file transfer is complete, is to test the return code from `api_block`. If `api_block` returns `0`, then Reflection has exhausted all queues. The `api_block` function also gives Reflection a little CPU time.

Combining Queued and Synchronous Commands

If you plan to use both queued and synchronous commands in your application, issue `api_docommand()` only after `api_startcommands()` and before `api_endcommands()`.

For example:

```
if( !api_startcommands() )      /* execute this code only if */
{                                /* api_startcommands() succeeds */
    ...
    api_docommand(...);
    ...
    api_endcommands();
}
```

An asynchronous command (`api_qcmd()`) issued from within the block defined in this example will fail.

If there are any pending, queued commands, `api_startcommands()` will fail.

If it is necessary to execute `api_docommand()` while there are pending commands in the queue, you have to give Reflection time to finish processing the queued commands. This can be accomplished in two ways:

Use `api_wait()`:

```
api_qcmd( "wait 0:0:30" );
api_wait();
if( !api_startcommands() )
{
    api_docommand( "display 'Your time is up!'" );
    api_endcommands();
}
```

Use `api_block()`:

```
api_qcmd( "wait 0:0:30" );
while( api_block() )
    if( kbhit() ) break;
if( !api_startcommands() )
{
    api_docommand( "display 'Your time is up!'" );
    api_endcommands();
}
```

The difference between these two approaches is that `api_wait()` does not give the API application the possibility of breaking out: it will wait until all of the queued commands are executed. With `api_block()` there is a way of getting out.

Timeouts

If you are using synchronous calls (`api_docommand`), where the API application has to wait for the call to complete, make sure that you either use the `api_setgetintstate` function, or specify a timeout on any WAIT, HOLD, READ-HOST, or other commands. Your application could hang waiting for a response that never arrives.

For example, if you send Reflection a command like `WAIT FOR "Main Menu"`, be very sure that the text "Main Menu" is going to come down from the host computer within a reasonable time period. Otherwise, the function call will never return, and the application will hang waiting for the required message to come from the VAX or HP 3000.

Rather than risk this, make sure that all commands have a timeout period:

```
api_docommand("WAIT 0:0:2 for 'Main Menu'");
```

This command waits up to 2 seconds for the string. If the string has already arrived or arrives in 5 seconds, the call returns quickly to the API application. Another API function lets you find out if the string was actually found: `api_found`.

A second way to get out of the infinite wait is to enable interruptibility with the `api_setgetintstate` function. This function enables users to break out of a wait by pressing **Ctrl-Break** during execution of a command. For example:

```
api_setgetintstate(1);
api_docommand("WAIT 0:0:30 for 'Main Menu'");
```

Here the `WAIT` command will either time out after 30 seconds or be interrupted by the user with **Ctrl-Break**.

Keyboard Reads

You will probably want to avoid `api_docommand("accept v1")`. `ACCEPT` is a Reflection keyword that reads input from the keyboard. Since Reflection is in background, the user won't be able to type anything (unless you have queued keys) and your program will hang.

You are probably accustomed to using the `C gets` command or some other variant for reading keyboard input into your C program. Unfortunately, many Microsoft C and Turbo Pascal keyboard reads use DOS calls like function 3FH (read file handle). This is similar to UNIX. When these calls are made to read keyboard input, they don't complete until you enter a whole line terminated by a carriage return. Since DOS is not re-entrant and only one process can make a DOS call at a time, calls of this type cause Reflection to freeze until they complete. No multitasking takes place. This can halt a file transfer and cause the host or Reflection to timeout. This should not be an issue unless you use queued commands.

To avoid this problem, you will need to use different calls to read from the keyboard. The libraries (on disk) include the calls `rgets` for C and `rfreadkeybd` for Pascal.

Conflicting Commands

When Reflection is running in the background, it is constantly accepting commands from both the host computer and the API application. These commands may occasionally conflict. For instance, the host may lock the keyboard just as the API application is sending keystrokes for the next screen.

The host can also invoke Reflection commands by prefixing Reflection commands with the escape sequence `ESC&OC`. Since some host programs use this facility, Reflection may get a command sent with this escape sequence while it is executing a command or command file begun by the user or the API application. Since Reflection can't process two commands at once, the host commands fail. Because the host program isn't behaving the same way it was when you ran it from the keyboard, the API application calls also fail.

The command `SET HOST-INITIATED-COMMANDS NO` turns off host-initiated commands: if any commands come from the host, they will fail. However, this still seriously affects or aborts the host application. There's no solution to problems of this kind except awareness. It is often helpful to make host commands, status requests, etc., visible so that you can see what your program has to take into account. Use `SET DISPLAY-CONTROL-CHARACTERS YES` to see what the host is actually doing.

Preventing User Exits

An unguarded API application can be unintentionally sabotaged by the user. If the user pops up Reflection and does a hard exit (`(Alt-X)`), your application will have nothing to interface with. The only way that your application can detect this is by looking at return codes. If you're programming in Assembler, `AX` returns `0DE00h` if Reflection has been uninstalled. In C or Pascal, the API function returns `-1`.

To make sure that the user can't abort Reflection, use the Reflection `SET` commands below. These settings make it more difficult to debug your program, so don't use them until the program is working correctly.

- ▲ SET HOT-KEY NONE

Prevents the user from popping up Reflection

- ▲ SET EXITS-DISABLED YES

Prevents the user from exiting Reflection ((Alt)-(X)) or doing a hard reset

- ▲ SET DISABLE-INTERRUPT YES

Prevents the user from stopping a command ((Ctrl)-(Y))

Uninstalling Reflection

If you need to uninstall Reflection to free up some memory, use `api_qcmd` as follows:

```
api_qcmd( "HARDEXIT" );
while ( !api_rcheck() )
    api_block();
```

Other methods may leave your PC in an unstable state.

Datacomm in `api_docommand` Sequences

Once the `api_startcommands` function is performed, datacomm is turned off except during the actual execution of an `api_docommand`. This is essential to allow your application to remain synchronized with the host program. When datacomm is turned off, Reflection will not read any incoming characters from its receive buffer unless a command is being processed. Characters are still received, but they remain in the buffer. Reflection ignores them.

This means that the API application can pause and prompt the user for something without missing any incoming characters. If the API application is logging a user onto a DEC VAX, for example, and has to stop and prompt for a password, there is no way for the password prompt to go undetected. The prompt stays in Reflection's receive buffer waiting to be read by the next command:

```
api_docommand( "WAIT 0:0:30 FOR 'Password:' " );
```


This command returns immediately since the prompt was already in the receive buffer.

If, instead, Reflection always read datacomm and the above wait command were issued, the WAIT would time out in 30 seconds without finding the 'Password:' prompt since it had already been received. The API application would assume that there was a problem communicating with the VAX and proceed accordingly.

To prevent Reflection's receive buffer from being overrun by the host during delays by the API application, you must set the appropriate type of flow control. For both HP and DEC, this will usually be XON/XOFF receive pacing.

When a Reflection WAIT FOR command is issued, it should be followed with an `api_found` command. This tells your application whether the WAIT actually found the string you were looking for, or whether it timed out without finding it.

```
api_docommand( "WAIT 0:0:30 FOR 'HP3000'" );
if ( api_found() )
    printf( "Logged on Successfully\n" );
else
    ...
```

Configuration Issues

The `api_getinfo` command returns a structure of configuration values that appear in configuration or setup dialog boxes but have no corresponding SET command. Most of these items will not affect your API application.

You can use the VALUE command to get more information on common configuration items such as baud rate, parity, etc. To find the baud rate, for instance, execute the following sequence:

```
api_docommand( "LET V3=VALUE(BAUD)" );
api_getvar( baudvalue, 3, length );
baudvalue[length]='\0';
printf( "Reflection baud rate = %s\n",baudvalue );
```

SET commands can be issued to change most configuration items. Some items cannot be changed except via SET commands or DISPLAY commands that use escape sequences.

Compiling and Linking with API Libraries

The following section gives instructions on building API applications with Borland Turbo Pascal, Microsoft QuickBASIC, Microsoft Professional BASIC, Microsoft C, and Borland C.

Building API Applications with Borland Turbo Pascal

Version 5.0 or greater of Borland Turbo Pascal is required.

Using a Turbo integrated environment:

1. Load Reflection in background using a minimum memory configuration.
2. Load SAMPLE.PAS file using pull-down menus.
3. Run SAMPLE.PAS. It will prompt for Reflection commands—use commands like `DIR`, `DISPLAY ^G` to sound the bell, or `FOREGROUND` to pop Reflection up.

Using the DOS command line:

- ▲ Invoke Turbo Pascal compiler via TPC command:

```
TPC sample /GD
```

The /GD switch produces a detailed map that can be used for debugging.

Turbo Pascal will “link” in APIUNIT.TPU to create SAMPLE.EXE.

Building API Applications with Microsoft QuickBASIC

Version 4.5 or greater of Microsoft QuickBASIC is required.

There are some special considerations for BASIC string variables. The size of a BASIC string variable cannot be modified by a called subroutine such as `api.getvar`. Any attempt to modify the length of a BASIC string variable results in a “string space corrupt” message and immediate termination of the program. If you need to get a Reflection variable into a BASIC string, you must first initialize the string to the correct length in BASIC and then pass the string to Reflection:

```
varbuf$ = SPACE$(80)
i% = api.getvar( varbuf$, 2, 1% )
varbuf$ = MID$( varbuf$, 1, 1% )
```

Using the QuickBASIC environment:

1. At the DOS prompt, load R1 or R2 using minimum memory configuration and with API support (/W). You must be using Reflection version 3.40 or greater. Reflection should immediately go into background if there is a configuration file.

```
R1 /W /B /I /Q /E
```

For switch definitions, type R1 /?.

2. Start the QB environment by typing QB <filename> /L B_APILIB.QLB at the DOS prompt, where <filename> is the name of the source file.

The QuickBASIC environment will start up with the API support present.

3. Load the SAMPLE.BAS program using the FILES pull-down menu.
4. Run the program. SAMPLE.BAS displays the Reflection serial number and then pops up Reflection after you press any key. Press the hot-key—[Alt]-(right)[Shift]—to return to the BASIC program.

Invoke the QuickBASIC compiler on the DOS command line:

```
BC SAMPLE;
```

Invoke the Microsoft linker, linking in the B_APILIB.LIB library:

```
LINK SAMPLE,,,B_APILIB;
```

There are a number of command line options for producing stand-alone .EXE versions and providing debug capabilities. These options are covered in the QuickBASIC documentation.

Building API Applications with Microsoft Professional BASIC

Microsoft Professional BASIC is required.

Make sure that BC.EXE and LINK.EXE are in your path or current directory, and that API.INC is in your current directory or the path specified by the INCLUDE environment variable. For proper operation it is necessary that Professional BASIC applications be compiled with the /Fs switch (to enable far strings) and linked with the PBAPILIB.LIB library. An example might be:

```
BC SAMPLE /O /Fs;
LINK SAMPLE,,,PBAPILIB C:\BC7\LIB\BCL70EFR;
```


SAMPLE.BAS is first compiled with the default stand-alone library (/O switch) and then linked with the PBAPILIB.LIB and BCL70EFR.LIB libraries. The latter is found in the C:\BC7\LIB directory. For further information on compiling and linking options, see your Professional BASIC documentation.

Building API Applications with Microsoft C

Microsoft C version 5.0 or greater is required.

Sample compile and link of a small model API application:

```
cl sample.c /link c_sapi.lib
```

Sample compile and link of a medium model API application:

```
cl /AM sample.c /link c_mapi.lib
```

Building API Applications with Borland C

Assuming that your "include" and API library files are in C:\API, compile and link a "sample.c" application program with the following options:

```
bcc -c -ms -Ic:\api sample.c  
tlink /Lc:\borlandc\lib;c:\api c0s sample,sample,,c_sapi cs
```

-c Compile only

-ms Small model

-I Include directory

-L Library directories

The Borland libraries are assumed to be in C:\BORLAND\LIB.

Converting Command Language to API

A good way to get started with API is to convert one or more command files to an API program. This chapter shows a short example in C using the `api_docommand()`, followed by a dialing program: first in Reflection command language, then in a C program using API calls.

One of the principal differences you'll notice when using the `api_docommand()` function is that the IF, ELSE, ENDIF, GOTO, and GOSUB commands are not supported. Flow control must be handled by the client program. (See page 244 for information on making a program interruptible.)

A command language IF clause might look like this:

```
CONTINUE
SEND "ABC TO XYZ"
IF ERROR
    DISPLAY "ERROR TRANSFERRING FILE^m^j"
ENDIF
```

This would be converted to C as follows:

```
; (CONTINUE is on automatically)
if ( api_docommand("SEND ABC TO XYZ") )
    printf( "Error transferring file\n" );
```

The `api_docommand()` returns 0 if there is no error, and the error code if an error occurred. This is then tested by C and a branch can be taken.

A BASIC version of the same thing would be as follows:

```
if api.docommand( "SEND ABC TO XYZ" ) > 0 then
    print "Error transferring file"
```

This command language example gets V1 from Reflection and then tests for a substring:

```
TRANSMIT "LISTF FOO^m"
READHOST V1
READHOST V1
WAIT 0:0:8 FOR "^q"
IF FIND("CIERR",v1) > 0
    DISPLAY "File FOO not found on host^m^j"
ENDIF
```

The C version of the same example is:

```
int    length;
char   varbuf[81];
api_docommand( "transmit 'listf foo^m'" );
api_docommand( "readhost 0:0:5 v1" );
api_docommand( "readhost 0:0:5 v1" );
api_docommand( "wait 0:0:8 for '^q'" );
api_getvar( varbuf, 1, &length );           /* get v1 */
varbuf[length] = '\0';                      /* null-terminate */
if ( strstr(varbuf, "CIERR") )
    printf( "File FOO not found on host\n" );
```

The `strstr()` function is the C counterpart of Reflection's `FIND` function.

Command Language Dialing Program

The following command file dials a modem and attempts to log on to an HP 3000 computer. Following it on page 276 is the same function in a C program using API calls.

```
SET DATACOMM-PORT COM1
SET BAUD 2400
SET CHARACTER-DELAY 80
;
; Need to set big delays and long waits when talking to modems
;
; Initialize retry_count
LET V1 = 0
DISPLAY 'Initializing modem...^M^J'
```



```

WAIT 0:0:1
TRANSMIT '+++ '
WAIT 0:0:2 FOR 'OK'
WAIT 0:0:.5
TRANSMIT 'ATH^M'
WAIT 0:0:2 FOR 'OK'
IF NOT FOUND
    LET V4 = 'No response from modem^M^J'
    GOTO FAIL
ENDIF
WAIT 0:0:1
DISPLAY 'Dialing modem...^M^J'
TRANSMIT 'ATDT 1234567^M'
WAIT 0:0:45 FOR 'CONNECT'
IF NOT FOUND
    LET V4 = 'Did not receive CONNECT message^M^J'
    GOTO FAIL
ENDIF
DISPLAY 'Connected to remote modem^M^J'
;
; Modem indicates connected - attempt to log on
:TRYAGAIN
IF V1 < 6
    LET V1 = V1 + 1
    TRANSMIT '^M'
    WAIT 0:0:1 FOR '^Q'
    IF NOT FOUND
        DISPLAY 'Try $1 failed^M^J'
        GOTO TRYAGAIN
    ELSE
        GOTO LOGON
    ENDIF
ELSE
    LET V4 = 'Never got host prompt^M^J'
    GOTO FAIL
ENDIF
DISPLAY 'Entering logon and password^M^J'
TRANSMIT 'hello doni,mgr.vers/joshua^M'
WAIT 0:0:30 FOR 'HP3000'

```

```

IF NOT FOUND
    DISPLAY 'Logon rejected - hang up modem^M^J'
    WAIT 0:0:2
    TRANSMIT '+++'
    WAIT 0:0:3 FOR 'OK'
    WAIT 0:0:1
    TRANSMIT 'ATH^M'
    WAIT 0:0:2 FOR 'OK'
    IF NOT_FOUND
        LET V4 = 'Hangup complete^M^J'
        GOTO FAIL
    ELSE
        LET V4 = 'Hangup failed^M^J'
        GOTO FAIL
    ENDIF
ENDIF
WAIT 0:0:30 FOR ':^Q'
SET CHARACTER-DELAY 0
DISPLAY 'Successful logon^M^J'
STOP
;
; Fail 'subroutine'
:FAIL
DISPLAY V4
STOP

```

C Version

The C version of the same command file using synchronous (do while you wait) commands follows. Notice that C is used to handle all of the logic and flow control rather than Reflection.

```

main()
{
    int    retry_count;
    char  serbuf[32];
    if ( api_instchk(serbuf) ) {
        printf( "API not present\n" );
        exit();
    }
}

```

```

if ( api_startcommands() ) {
    printf( "API busy\n" );
    exit();
}

api_docommand( "alert 'API - AUTODIAL - LOGON'" );
api_docommand( "set datacomm-port com1" );
api_docommand( "set baud 2400" );
api_docommand( "set character-delay 80" );

/* Set big delays and long waits when talking to modems */
retry_count = 0;
printf( "Initializing modem...\n" );
api_docommand( "wait 0:0:1" );
api_docommand( "transmit '+++' );
api_docommand( "wait 0:0:2 for 'OK'" );
api_docommand( "wait 0:0:.5" );
api_docommand( "transmit 'ATH^m'" );
api_docommand( "wait 0:0:2 for 'OK'" );
if ( !api_found() )
    fail( "No response from modem\n" );

api_docommand( "wait 0:0:1" );
printf( "Dialing modem...\n" );
api_docommand( "transmit 'ATDT 1234567^m'" );
api_docommand( "wait 0:0:45 for 'CONNECT'" );
if ( !api_found() )
    fail( "Did not receive CONNECT message\n" );

printf( "Connected to remote modem\n" );

/* Modem says we're connected - see if we can log on */
for ( retry_count = 0 ; retry_count < 6 ; retry_count++ ) {
    api_docommand( "transmit '^m'" );
    api_docommand( "wait 0:0:1 for '^q'" );
    if ( !api_found() )
        printf( "Try %d failed\n", retry_count );
    else
        break;
}

```



```

if ( retry_count >= 6 )
    fail( "Never got host prompt\n" );
else
    printf( "Entering logon and password\n" );

api_docommand( "transmit 'hello doni,mgr.vers/joshua^m'" );
api_docommand( "wait 0:0:30 for 'HP3000'" );
if ( !api_found() ) {
    printf( "Logon rejected - hang up modem\n" );
    api_docommand( "wait 0:0:2" );
    api_docommand( "transmit '+++' );
    api_docommand( "wait 0:0:3 for 'OK'" );
    api_docommand( "wait 0:0:1" );
    api_docommand( "transmit 'ATH^m'" );
    api_docommand( "wait 0:0:2 for 'OK'" );
    if ( !api_found() )
        fail( "Hangup complete\n" );
    else
        fail( "Hangup failed\n" );
}
api_docommand( "wait 0:0:30 for ':^q'" );
api_docommand( "set character-delay 0" );
printf( "Successful logon\n" );
api_endcommands();
return 0;
}

fail(s)
char *s;
{
    printf( "%s",s );
    api_endcommands();
    exit();
}

```

Queuing Keystrokes

Be careful when using single and double quotes with the `api_qkeys` function. When passing keystrokes, keys are either passed as *function names*, (the same names used in Reflection's keyboard remapping) or as *quoted strings*. Function names such as `HARD-RESET`, `HOST-BREAK`, `RETURN`, `F1`, and `F2` are not put in quotes. Keystrokes passed as quoted strings must be in quotes.

The following passes the function name `RETURN` as a non-quoted string to API—it is the same as pressing :

```
api_qkeys( "RETURN" );
```

The example below passes the keystrokes , , , , , and  to the API application.

```
api_qkeys( "'return'" );
```




Index

A

- Alphanumeric
 - mode 148
 - mode, status 153
- ANSI emulation
 - creating a keyboard mapping file 17
 - default mapping 40
- ANSI terminal
 - specifying 17
- api_assertdc 170
- api_atrbscreen 171
- api_block 173
- api_cancelbuf 175
- api_clrcmdq 176
- api_clrkeybd 178
- api_cmdstatus 180
- api_docommand 182
- api_endcommands 185
- api_found 186
- api_getfkey 188
- api_getinfo 192
- api_getstatus 198
- api_getvar 202
- API.INC
 - "include" file 162
- api_instchk 206
- api_keystatus 209
- api_ndcpopup 211
- api_offerbuf 213
- api_popup 216
- api_qcmd 218
- api_rcheck 223
- api_rdchar 226
- api_releasedc 230
- api_reset 232
- api_screenread 234
- api_searchscreen 238
- api_setfkey 242
- api_setgetintstate 244
- api_setvar 247
- api_startcommands 250
- api_wait 253
- api_writeasync 255
- api_xmitstatus 257
- Application Program Interface 157
 - asynchronous commands 262
 - converting Reflection command language 273
 - error codes 259, 262
 - function calls 258
 - getting configuration info 268
 - how to queue keystrokes 279
 - libraries 161
 - preventing a hard exit 266
 - queued vs. synchronous commands 261
 - reading keyboard input 265
 - Reflection loading switch 161
 - specifying a timeout 264
 - summary of function calls 165-167
 - support files 161
 - synchronizing with datacomm 267
 - synchronous commands 261
 - turning off host-initiated commands 266
 - what you need to know 261

B

- Backslash
 - in character string 30
- BBS
 - phone number ii
- Beam
 - use of 148
- Bitmap planes
 - ReGIS selection 115
- Break
 - keystroke, Tektronix 150
- Building an API application
 - Borland C 271
 - Borland Turbo Pascal 269

- Microsoft Professional BASIC 270
- Microsoft QuickBASIC 269
- Bulletin Board
 - phone number ii
- Bypass condition
 - description 153

C

- CapsLock
 - restrictions on mapping 25
 - setting initial state 18
- Caret
 - in a character string 30
- Case sensitivity
 - keyboard mapping 13
- Character cell, defining in ReGIS 134
- Character sets
 - creating ReGIS characters 134
 - loading in ReGIS 133–136
 - ReGIS selection 123
 - reporting, ReGIS 138
- Character size
 - ReGIS text 121, 126–128
- Character spacing, ReGIS text 125
- Character strings
 - concatenating 31
 - literal escape characters 30
 - special characters 30
- Circles
 - filling 118
- Clearing
 - graphics screen 100
- Color
 - defining in ReGIS 97
 - linked to ReGIS map 95
 - ReGIS background 95
 - ReGIS foreground 106
 - ReGIS specifiers 95
 - selecting ReGIS background 100
- Color graphics 96
- Command language
 - mapping to keystroke 32
- Command line
 - keyboard function 32

- Comments
 - in keyboard mapping 34
- Compiler
 - error messages 44
- Complement writing 109
- Concatenating
 - character strings 31
- Configuration files
 - returning to default 44
- Control characters
 - within ReGIS commands 77
- Cursor
 - graphics 101
 - position request, ReGIS 137
 - ReGIS position reporting 139, 140
 - user-defined ReGIS 102
- Cursor keypad
 - key names 24
- Curve command
 - arc with center at current position 88
 - arc with center at specified position 89
 - center of circle at current position 87
 - center of circle at specified position 88
 - closed curve sequence 90
 - open curve sequence 91
 - temporary write control 92
- Curves
 - filling 118

D

- Default configuration file 44
- Defining keystrokes
 - samples 6
- Digital
 - LK250 keyboard 13
 - LK450 keyboard 13
 - LK250 key names 53
- Display addressing, ReGIS 93
- Display cell size 126
- Distortion of ReGIS text 128
- Drawing
 - patterns 103–116

E

- Endpoints 148
 - plotting 155
- Enhanced keyboard 13, 52
- Enter/exit Tektronix
 - keystroke 150
- Erase
 - display, keystroke 150
- Erase writing 110
- Erasing
 - graphics screen 100
- Error codes
 - API 259
- Error messages
 - KEYCOMP 44
- Errors
 - requesting from ReGIS 138
- Esc character
 - representing in strings 29

F

- Fill command
 - see Polygon fill command 117
- Filling polygons 117–119
- Form feed
 - after printing 152
- Function calls
 - summary 165–167
- Function keys 25

G

- GIN mode 148
 - status 153
- Graphics
 - color 96
 - cursors 101
 - display mode, Tektronix 148
 - input mode 139
 - mode, entering and exiting ReGIS 69
 - monochrome 96
 - plot mode status 153

- printing 94
- ReGIS 144
- ReGIS color selection 106
- ReGIS command notation 73
- ReGIS pattern control 103–116
- screen command 93
- text 121–132

H

- Help system
 - custom keys 10
 - Keystrokes help 10
- HLS color specifiers 95, 106
- HP Vectra 13
 - keyboard 54

I

- IBM
 - Enhanced keyboard 52
 - PC/AT 13
 - PC/AT keyboard 51
 - PC or PC/XT 13
 - PC/XT keyboard 50
- Input cursor 101–102
- Input specification
 - key names 21
- Interrupt 16
 - keyboard mapping 60
- Interrupt 9
 - keyboard mapping 59
- Interrupt values
 - keyboard 58
- Italics, ReGIS text 130

K

- KBM files 8
- Keyboard
 - default mapping 10, 34
 - information 14
 - interrupt values 58
 - mapping VT functions 8

- names 13
 - supplied mappings 8
 - WordPerfect mapping file 9
 - Keyboard functions
 - command line 32
 - Keyboard mapping
 - alphabetic key names 21
 - case sensitivity 13
 - comments, in keystroke definitions 34
 - compiling 43
 - configuration file 43
 - configuration file, default 44
 - cursor keypad key names 24
 - default keystrokes 34
 - defining actions (output) 26
 - defining keystrokes (input) 19
 - function keys 25
 - global 5
 - help system 10
 - identifying keyboard 13-17
 - identifying terminal type 17
 - initializing keyboard 18
 - input specification 19-25
 - keyboard function names 34
 - keyboard layouts 49-55
 - KEYCOMP compiler 43
 - KEYCOMP error messages 44
 - Shift, Ctrl, Alt 20
 - menu shortcut keys 24
 - numeric keypad key names 22
 - output specification 26-33
 - overview 1
 - Reflection keyboard functions 34
 - reset functions 27
 - sample definitions 6
 - shift keys 20
 - special default keystrokes 24
 - syntax 13-34
 - syntax, quick reference 12
 - terminal mode 5
 - VT control functions 32
 - VT functions 8
 - VT terminal 35
 - KEYCOMP.EXE
 - error messages 44
 - mapping compiler 43
 - mapping walk-through 8
 - Key definitions
 - special characters 30
 - syntax 19
 - KEYMON.COM 57
 - Key names in mapping files
 - Digital LK250 keyboard 53
 - HP Vectra keyboard 54
 - IBM Enhanced keyboard 52
 - Key Tronic 5151 keyboard 55
 - PC/AT keyboard 51
 - PC/XT keyboard 50
 - Keystroke
 - definition syntax 19
 - detecting 58
 - high byte 58
 - low byte 58
 - mapping 19
 - scan code 58
 - Keystrokes help 10
 - Key Tronic
 - KB 5151 keyboard 13, 55
- ## L
- Literal escape 30
 - LK250
 - keyboard 13
 - LK450
 - keyboard 13
 - Load command 133-136
 - defining characters 134
 - naming character set 133
 - selecting a character set 133
 - Loading
 - Tektronix 147
 - Local
 - and online toggle 150
 - Lotus 1-2-3
 - keyboard mapping for VAX/VMS program 9

M

- Macrograph command 143–144
- Macrograph reporting 137
- Mapping
 - keyboard 5
- Margin
 - Tektronix, description 148
- MC
 - graphics printing 94
- Monochrome graphics 96
- Mouse
 - buttons, default mapping 71
 - ReGIS button codes 141
 - support, Reflection 4 71
- Mouse support
 - mapping buttons 39
- Move
 - cursor, Tektronix 151
 - window, Tektronix 151
- Multiple graphics input mode 139

N

- National replacement character sets
 - selecting for ReGIS text 124
- Numeric keypad
 - cursor key names 22
 - numeric key names 22
- NumLock
 - restrictions on mapping 25
 - setting initial state 18

O

- One-shot graphics input mode 139
- Output cursor 101
- Output mapping
 - ReGIS background colors 95
 - ReGIS foreground colors 106
- Overlay writing 107
- Overstriking ReGIS text 132

P

- Page
 - Tektronix function 151
- Patterns, ReGIS write command 104
- PC
 - keyboard 50
- Phone numbers
 - Walker Richer & Quinn ii
- Pixel vectors
 - positioning text 132
- Plane selection, ReGIS 115
- Plotting graphics data 155
- Polygon fill command 117–119
 - curves/circles 118
 - specifying coordinates 117
 - temporary write control 119
- Position command
 - begin a bounded position stack 81
 - begin an unbounded position stack 81
 - cursor positioning 79
 - cursor positioning using pixel vectors 80
 - multiplication of PV values 80
 - null position argument 82
- Print
 - buffer 152
 - screen, graphics 152
 - screen, keystroke 151
- Printing
 - ReGIS commands 94
- Print screen
 - graphics 94
- Problems
 - correcting distorted ReGIS text 128
- Programmatic control of Reflection 157

Q

- qkeys 220
- Quote marks
 - in keyboard remapping 30
 - rules for using 33

R

- Reflection 4
 - loading 69
 - mouse support 39
 - sample session 70
- Reflection's command line
 - keystroke 151
- ReGIS 144
 - character sets, defining 133
 - command summary 74
 - command syntax 73-77
 - curve command 87
 - defining colors 97
 - definition 63
 - erasing screen 100
 - error reporting 138
 - load command 133-136
 - macrograph command 143-144
 - notational conventions 73
 - pixels 78
 - pixel vector directions 78
 - pixel vector multiplication 78
 - plane selection 115
 - polygon fill command 117-119
 - position command 79
 - printing 94
 - report command 137-141
 - screen command 93-102
 - text character selection 123
 - text command 121-132
 - vector command 83
 - write command 103-116
 - writing styles 107
- Remapping
 - keyboard 5
 - mouse buttons, Reflection 4 71
- Replace writing 108
- Report command 137-141
 - character set 138
 - errors 138
 - graphics input modes 139
 - interactive cursor position 140
 - macrograph contents 137

- Request of ReGIS cursor position 140
- Reset
 - keystroke, Tektronix 151
 - restrictions on mapping 27
- RGB color specifiers 106

S

- Sample keystroke definitions 6
- Sample session
 - Reflection 4 70
 - Tektronix emulation 149
- Scaled
 - and unscaled toggle, Tektronix 152
- Scan code
 - generated by key 58
- Screen
 - dump, Tektronix 152
- Screen command 93-102
 - background color 100
 - color graphics 96
 - erase 100
 - graphics cursors 101
 - monochrome graphics 96
 - output mapping 95
 - printing graphics 94
 - scrolling 93
 - time delay 100
- SET
 - GLOBAL-REMAPPING 5
 - TERMINAL-TYPE 17
- Set alpha mode
 - keystroke 150, 152
- Shading
 - ReGIS images 110
 - with characters 111
- Shift keys
 - restrictions 20
- Sixel
 - print destination, ReGIS screen command 94
- Spacing between ReGIS text 125
- Special characters
 - in character strings 30
- Speed cursor
 - keystroke 152

- Status, ReGIS reports 137
- Subscripting ReGIS text 132
- Superscripting ReGIS text 132
- Supplied mapping files 43
 - for WordPerfect users 9
- Support files
 - Application Program Interface 161
- Syntax
 - keyboard mapping 13, 19

T

- Technical support
 - phone number ii
- Tektronix
 - graphics 147
 - switching to 147
 - terminal status request 153
- Terminal
 - status request, Tektronix 153
 - status, Tektronix 148
- Terminal mode
 - keyboard mapping 5
- Terminal type identification
 - keyboard mapping 17
- Text command 121–132
 - character set selection 123
 - character size 126–128
 - character spacing 125
 - height multiplier 127
 - italics 130
 - notation 122
 - positioning text 132
 - size multiplier 128
 - string tilt 128
 - temporary text control 131
 - temporary write control 131
- Text, in ReGIS graphics 121
- Toggle
 - text and graphics modes 69
- TYPE
 - testing a ReGIS program 70

U

- Unit cell size 126

V

- Vector command
 - begin bounded position stack 84
 - begin unbounded position stack 85
 - drawing a dot 83
 - drawing a line 83
 - drawing a line using pixel vector system 84
 - temporary write control 85
- Vectra keyboard 54
- VT241
 - color mapping 99
- VT340
 - color mapping 99
- VT control functions
 - mapping to keystroke 32
- VT terminal
 - specifying 18

W

- Walker Richer & Quinn
 - phone numbers ii
- WordPerfect
 - keyboard mapping 9
- Write command 103–116'
 - examples 112–114
 - foreground color 106
 - negative pattern control 110
 - pattern selection 104–105
 - plane selection 115
 - polygon fill, temporary control 119
 - shading 110
 - shading with characters 111
 - text, temporary control 131
 - writing styles 107–110





